Java
Полное руководство
12-е издание
Всестороннее раскрытие языка Java

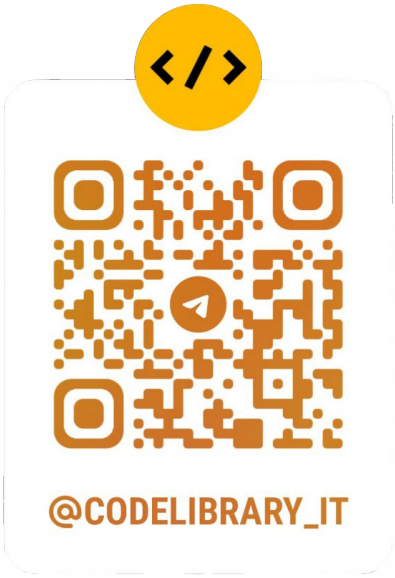
Полностью обновленное и расширенное издание

Герберт Шилдт

Mc Graw Hill Education

ДИАЛЕКТИКА

Полное руководство Java 12-е издание



The Complete Reference

Java™

Twelfth Edition

Herbert Schildt

New York Chicago San Francisco
Athens London Madrid Mexico City
Milan New Delhi Singapore Sydney Toronto

Полное руководство Java 12-е издание

Герберт Шилдт

Москва • Санкт-Петербург
2022

ББК 32.973.26-018.2.75
Ш57
УДК 004.432

ООО "Диалектика"

Перевод с английского и редакция *Ю.Н. Артеменко*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info.dialektika@gmail.com, http://www.dialektika.com

Шилдт, Герберт.

Ш57 Java. Полное руководство, 12-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2023. — 1344 с. : ил. — Парал. тит. англ.

ISBN 978-5-907458-86-4 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Герберт Шилдт
Java. Полное руководство
12-е издание

Подписано в печать 12.09.2022. Формат 70×100/16
Усл. печ. л. 108,36. Уч.-изд. л. 68,2
Тираж 500 экз. Заказ № 7327

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907458-86-4 (рус.)

ISBN 978-1-26-046341-5 (англ.)

Оглавление

Содержание

Об авторе

Автор многочисленных бестселлеров **Герберт Шилдт** занимался написанием книг по программированию на протяжении более трех десятилетий и считается признанным авторитетом по языку Java. Журнал *International Developer* назвал его одним из ведущих авторов книг программированию в мире. Книги Герберта Шилдта продавались миллионными тиражами по всему миру и переведены на многие языки. Он является автором многочисленных книг по Java, в том числе *Java: руководство для начинающих*, *Java: методики программирования Шилдта*, *Introducing JavaFX 8 Programming* и *Swing: руководство для начинающих*. Герберт Шилдт также много писал о языках C, C++ и C#. В книге Эда Бернса *Secrets of the Rock Star Programmers: Riding the IT Crest* указано, что как один из звездных программистов Шилдт интересуется всеми аспектами вычислений, но его основное внимание сосредоточено на языках программирования. Герберт Шилдт получил степени бакалавра и магистра в Иллинойском университете. Его сайт доступен по адресу `www.HerbSchildt.com`.

О научном редакторе

Доктор **Дэнни Ковард** работал над всеми версиями платформы Java. Он руководил определением сервлетов Java в первой версии платформы Java EE и в более поздних версиях, внедрением веб-служб в Java ME, а также стратегией и планированием для Java SE 7. Он основал технологию JavaFX и совсем недавно создал крупнейшее дополнение к стандарту Java EE 7, Java WebSocket API. Доктор Дэнни Ковард обладает уникально широким взглядом на многие аспекты технологии Java. Его опыт простирается от написания кода на Java до разработки API с отраслевыми экспертами и работы в течение нескольких лет в качестве исполнительного директора в Java Community Process. Кроме того, он является автором двух книг по программированию на Java: *Java WebSocket Programming* и *Java EE 7: The Big Picture*. Совсем недавно он применил свои знания Java, помогая масштабировать крупные службы на основе Java для одной из самых успешных в мире компаний-разработчиков программного обеспечения. Доктор Дэнни Ковард получил степень бакалавра, магистра и доктора математики в Оксфордском университете.

Предисловие

Java — один из самых важных и широко используемых языков программирования в мире. На протяжении многих лет ему была присуща эта отличительная особенность. В отличие от ряда других языков программирования, влияние которых с течением времени ослабевало, влияние Java становилось только сильнее. С момента своего первого выпуска язык Java выдвинулся на передний край программирования для Интернета. Его позиции закреплялись с каждой последующей версией. На сегодняшний день Java по-прежнему является первым и лучшим выбором для разработки веб-приложений, а также мощным языком программирования общего назначения, подходящий для самых разных целей. Проще говоря, большая часть современного кода написана на Java. Язык Java действительно настолько важен.

Ключевая причина успеха языка Java кроется в его гибкости. С момента своего первоначального выпуска 1.0 он постоянно адаптировался к изменениям в среде программирования и к изменениям в способах написания кода программистами. Самое главное то, что язык Java не просто следовал тенденциям — он помогал их создавать. Способность языка Java приспосабливаться к быстрым изменениям в мире программирования является важной частью того, почему он был и остается настолько успешным.

С момента первой публикации этой книги в 1996 году она выдержала множество переизданий, в каждом из которых отражалась непрерывная эволюция Java. Текущее двенадцатое издание книги обновлено с учетом Java SE 17 (JDK 17). В итоге оно содержит значительный объем нового материала, обновлений и изменений. Особый интерес представляет обсуждение следующих ключевых возможностей, которые были добавлены в язык Java в сравнении с предыдущим изданием:

- усовершенствования оператора `switch`;
- записи;
- сопоставление с образцом в `instanceof`;
- запечатанные классы и интерфейсы;
- текстовые блоки.

В совокупности они составляют существенный набор новых функциональных средств, которые значительно расширяют диапазон охвата, область применимости и выразительность языка. Усовершенствования `switch` добавляют мощи и гибкости этому основополагающему оператору управления. Появившиеся записи предлагают эффективный способ агрегирования данных. Добавление сопоставления с образцом в `instanceof` обеспечивает более рациональный и устойчивый подход к решению обычной задачи программирования. Запечатанные классы и интерфейсы делают возможным детализированный контроль над наследованием. Текстовые блоки позволяют вводить многострочные строковые литералы, что значительно упрощает процесс вставки таких строк в исходный код. Все вместе новые функциональные средства существенно расширяют возможности разработки и внедрения решений.

Книга для всех программистов

Книга предназначена для всех программистов: от новичков до профессионалов. Новичок сочтет особенно полезными тщательно продуманные обсуждения и множество примеров. Профессионалам понравится подробное описание более сложных функциональных средств и библиотек Java. И те, и другие получат в свое распоряжение прочный информационный ресурс и удобный справочник.

Структура книги

Эта книга представляет собой исчерпывающее руководство по языку Java, в котором описаны его синтаксис, ключевые слова и базовые принципы программирования. Вдобавок исследуется значительное подмножество библиотеки Java API. Книга разделена на четыре части, каждая из которых посвящена отдельному аспекту среды программирования Java.

Часть I предлагает подробное учебное пособие по языку Java. Она начинается с основ, включая типы данных, операции, управляющие операторы и классы. Затем обсуждаются наследование, пакеты, интерфейсы, обработка исключений и многопоточность. Далее описаны аннотации, перечисления, автоупаковка, обобщения, модули и лямбда-выражения. Также предлагается введение в систему ввода-вывода. В последней главе части I рассматривается несколько недавно добавленных средств: записи, запечатанные классы и интерфейсы, расширенный оператор `switch`, сопоставление с образцом в `instanceof` и текстовые блоки.

В части II исследуются ключевые аспекты стандартной библиотеки API Java, включая строки, ввод-вывод, работу в сети, стандартные утилиты, Collections Framework, AWT, обработку событий, визуализацию, параллелизм (в том числе Fork/Join Framework), регулярные выражения и библиотеку потоков.

В трех главах части III дается введение в инфраструктуру Swing.

Часть IV состоит из двух глав, в которых демонстрируются примеры приложений Java. В одной главе обсуждается компоненты Java Beans, а в другой представлено введение в сервлеты.

Благодарности

Я хочу выразить особую благодарность Патрику Нотону, Джо О'Нилу и Дэнни Коварду.

Патрик Нотон был одним из создателей языка Java. Он также помог написать первое издание этой книги. Например, помимо многих других вкладов большая часть материала в главах 22, 24 и 28 изначально была предоставлена Патриком. Его проницательность, опыт и энергия во многом способствовали успеху книги.

Во время подготовки второго и третьего изданий этой книги Джо О'Нил предоставил первоначальные наброски материала, который сейчас содержится в главах 31, 33, 35 и 36 текущего издания. Джо помогал мне при написании нескольких книг, и его вклад всегда был превосходным.

Дэнни Ковард занимался научным редактированием этого издания книги. Дэнни работал над несколькими моими книгами, и его советы, идеи и предложения всегда имели большую ценность и получали высокую оценку.

Я также хочу поблагодарить мою жену Шерри за весь ее вклад в эту и в другие мои книги. Ее вдумчивые советы, корректура и подготовка предметного указателя всегда в значительной степени способствовали успешному завершению каждого проекта.

Герберт Шилдт

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info.dialektika@gmail.com`
WWW: `http://www.williamspublishing.com`

ЧАСТЬ

I Язык Java

ГЛАВА

1 История и эволюция языка Java

Глубокое понимание языка Java предусматривает знание причин его создания, факторов, придавших ему такую форму, и особенностей, которые он унаследовал. Как и предшествующие успешные языки программирования, Java представляет собой смесь наилучших элементов своего богатого наследия в сочетании с новаторскими концепциями, необходимыми для достижения его уникальной цели. В остальных главах книги описаны практические аспекты Java, включая синтаксис, основные библиотеки и приложения, а в этой главе обсуждаются причины появления языка Java, его важность и развитие с течением лет.

Хотя язык Java неразрывно связан с онлайновой средой Интернета, важно помнить о том, что Java в первую очередь является языком программирования. Развитие и создание языков программирования происходят по двум фундаментальным причинам:

- адаптация к меняющимся средам и сценариям использования;
- внедрение улучшений и усовершенствований в область программирования.

Как вы увидите, создание Java было обусловлено обеими причинами почти в равной степени.

Происхождение Java

Язык Java связан с языком C++ — прямым потомком C. Большинство характерных особенностей Java унаследовано от упомянутых двух языков. От C язык Java получил синтаксис. На многие средства объектно-ориентированного программирования Java повлиял язык C++. На самом деле некоторые определяющие характеристики Java происходят от предшественников или являются ответом на них. Кроме того, создание Java уходит глубоко в процесс улучшения и адаптации, который происходил в языках программирования на протяжении нескольких прошедших десятилетий. По указанным причинам в этом разделе рассматривается последовательность событий и движущих сил, которые привели к Java. Вы увидите, что каждое нововведение в конструкции

языка было обусловлено необходимостью решения фундаментальной задачи, которую нельзя было решить с помощью существовавших ранее языков. Язык Java — не исключение.

Зарождение современного программирования: язык C

Язык C серьезно всколыхнул компьютерный мир. Влияние языка C не следует недооценивать, поскольку он коренным образом изменил подход к программированию и его понимание. Создание C было прямым результатом потребности в структурированном, эффективном, высокоуровневом языке, который смог бы заменить ассемблерный код при написании системных программ. Вероятно, вам известно, что при проектировании языка программирования часто идут на компромиссы, такие как:

- простота использования или возможности;
- безопасность или эффективность;
- жесткость или расширяемость.

До появления C программистам обычно приходилось выбирать между языками, которые оптимизировали один или другой набор характеристик. Например, хотя язык FORTRAN можно было применять при создании довольно эффективных программ для научных расчетов, он не особо хорошо подходил для написания системного кода. Несмотря на простоту изучения языка BASIC, он не обладал широкими возможностями, а отсутствие структурированности ставило под сомнение его пригодность для создания крупных программ. Язык ассемблера можно было задействовать при построении высокопроизводительных программ, но изучать и эффективно использовать его нелегко. Вдобавок отладка ассемблерного кода может оказаться достаточно непростой.

Еще одна сложная проблема заключалась в том, что ранние языки программирования наподобие BASIC, COBOL и FORTRAN не проектировались с учетом принципов структурирования. Взамен они полагались на оператор GOTO как на главное средство управления потоком выполнения программы. В результате возникла тенденция появления в программах, написанных на таких языках, так называемого "спагетти-кода" — массы запутанных переходов или условных ответвлений, из-за которых программы было практически невозможно понять. Хотя языки вроде Pascal являются структурированными, они не разрабатывались для обеспечения эффективности и не имели определенных функций, необходимых в целях их применимости в широком диапазоне программ. (В частности, учитывая доступные в то время стандартные диалекты языка Pascal, не имело смысла рассматривать его использование для написания кода системного уровня.)

Итак, незадолго до изобретения C, невзирая на все прилагаемые усилия, ни один существующий язык не сглаживал конфликт между противоречивыми характеристиками. Однако потребность в таком языке была насущной.

К началу 1970-х годов компьютерная революция стала набирать обороты, и спрос на программное обеспечение быстро опередил возможности программистов создавать его. В академических кругах прилагалось много усилий для проектирования лучшего языка программирования. Но, пожалуй, важнее всего то, что начала воздействовать дополнительная сила. Компьютерное оборудование наконец-то стало настолько распространенным, что была достигнута критическая масса. Компьютеры больше не находились за запертыми дверями. Впервые программисты получили фактически неограниченный доступ к своим машинам, что дало свободу для экспериментирования. Кроме того, это позволило программистам начать разработку собственных инструментов. Накануне создания C была подготовлена почва для качественного скачка в области языков программирования.

Изобретенный и первоначально реализованный Деннисом Ритчи на машине DEC PDP-11 под управлением операционной системы (ОС) UNIX язык C стал результатом процесса разработки, который начался с более старого языка под названием BCPL, созданного Мартином Ричардсом. Язык BCPL оказал влияние на язык B, изобретенный Кеном Томпсоном, который привел к разработке C в 1970-х годах. В течение долгих лет фактическим стандартом языка C служила версия, которая поставлялась вместе с ОС UNIX и описана во втором издании книги Брайана Кернигана и Денниса Ритчи *Язык программирования C* (ИД "Вильямс", 2006 год). Язык C был официально стандартизирован в декабре 1989 года, когда Американский национальный институт стандартов (American National Standards Institute — ANSI) принял стандарт для C.

Многие считают, что создание языка C положило начало современной эпохи языков программирования. Он успешно объединил в себе противоречивые характеристики, которые представляли собой источник немалого беспокойства в более ранних языках. В результате появился мощный, эффективный, структурированный язык, который было относительно легко изучать. Ему также присуща еще одна почти неуловимая особенность: он являлся языком для программистов. До изобретения C языки программирования обычно разрабатывались либо в виде академических упражнений, либо бюрократическими организациями. Другое дело язык C. Он проектировался, реализовывался и развивался практикующими программистами, отражая их подход к программированию. Его функциональные средства были отточены, протестированы, продуманы и переосмыслены людьми, которые действительно использовали язык. Итогом стал язык, с которым программистам нравится работать. Более того, язык C быстро приобрел многочисленных приверженцев, которые демонстрировали почти религиозный пыл по отношению к нему. Таким образом, он нашел широкое признание в сообществе программистов. Выражаясь кратко, C представляет собой язык, спроектированный программистами и предназначенный для программистов. Как вы увидите, язык Java унаследовал эту традицию.

C++: следующий шаг

За период с конца 1970-х до начала 1980-х годов C стал доминирующим языком программирования и продолжает широко применяться в наши дни. Поскольку C является успешным и удобным языком, у вас может возникнуть вопрос: почему существует потребность в чем-то еще? Ответ — сложность. На протяжении всей истории программирования возрастающая сложность программ вызывала необходимость в поиске лучших способов управления такой сложностью. Язык C++ стал реакцией на эту потребность. Чтобы лучше понять причины, по которым управление сложностью крайне важно для создания C++, ознакомьтесь со следующими рассуждениями.

С момента изобретения компьютера подходы к программированию значительно изменились. Например, когда компьютеры только появились, программирование выглядело как ручное переключение двоичных машинных инструкций на передней панели компьютера. Пока программы состояли из нескольких сотен инструкций, такой подход был приемлемым. С ростом размера программ был изобретен язык ассемблера, чтобы программист мог работать с более крупными и постоянно усложнявшимися программами, применяя символические представления машинных инструкций. По мере того как размер программ продолжил увеличиваться, появились языки высокого уровня, которые предоставили программистам больше инструментов, предназначенных для управления сложностью.

Первым широко распространенным языком был, конечно же, FORTRAN. Хотя FORTRAN стал впечатляющим первым шагом, в те времена его трудно было назвать языком, способствующим созданию ясных и простых для понимания программ. В 1960-х годах зародилось структурное программирование. Эта методика программирования воплотилась в таких языках, как C. За счет использования структурированных языков у программистов впервые появилась возможность довольно легко писать умеренно сложные программы. Тем не менее, даже в случае применения методов структурного программирования при достижении проектом определенного размера его сложность превышала тот порог, с которым мог справиться программист. К началу 1980-х годов многие проекты вышли за рамки возможностей структурного программирования. Для решения проблемы был изобретен новый способ программирования, названный объектно-ориентированным программированием (ООП). Объектно-ориентированное программирование подробно обсуждается далее в книге, а пока вот его краткое определение: ООП — это программная методология, которая помогает приводить в порядок сложные программы с использованием наследования, инкапсуляции и полиморфизма.

В конечном итоге, хотя C является одним из великолепных языков программирования, его способность управлять сложностью имеет предел. После того как размер программы превышает определенную величину, она становится настолько сложной, что ее трудно воспринимать как единое целое. Несмотря на то что точный размер, при котором это происходит, отличается в зависимости от природы программы и программиста, всегда существует

порог, по достижении которого программа становится неуправляемой. В C++ были добавлены средства, позволяющие преодолеть такой порог, что позволило программистам понимать и управлять более крупными программами.

Язык C++ был изобретен Бьярне Страуструпом в 1979 году, когда он работал в Bell Laboratories (Мюррей Хилл, Нью-Джерси). Первоначально Страуструп назвал новый язык "C with Classes" (C с классами). Однако в 1983 году название изменилось на C++. Язык C++ расширяет C за счет добавления объектно-ориентированных возможностей. Поскольку язык C++ построен на основе C, он поддерживает все возможности, характерные черты и преимущества C. Это важнейшая причина успеха C++ как языка. Изобретение C++ не было попыткой создать совершенно новый язык программирования. Наоборот, он задумывался как усовершенствование уже крайне успешного языка.

Условия для появления языка Java

В конце 1980-х и начале 1990-х годов ООП с использованием C++ обрело популярность. Действительно, на мгновение показалось, что программисты наконец-то нашли идеальный язык. Так как язык C++ сочетал в себе высокую эффективность и стилистические элементы C с объектно-ориентированной парадигмой, его можно было применять для создания широкого спектра программ. Тем не менее, как и в прошлом, назревали силы, которые снова подтолкнули вперед эволюцию языков программирования. Через несколько лет Всемирная паутина и Интернет достигнут критической массы. Это событие ускорит еще одну революцию в программировании.

Создание языка Java

Язык Java придумали Джеймс Гослинг, Патрик Нотон, Крис Уарт, Эд Франк и Майк Шеридан из компании Microsystems, Inc. в 1991 году. Разработка первой рабочей версии заняла полтора года. Первоначально язык получил название Oak, но в 1995 году был переименован в Java. Между исходной реализацией Oak осенью 1992 года и публичным объявлением о Java весной 1995 года многие люди внесли свой вклад в проектирование и эволюцию языка. Так, Билл Джой, Артур ван Хофф, Джонатан Пэйн, Френк Йеллин и Тим Линдхольм сыграли ключевую роль в совершенствовании самого первого прототипа.

В какой-то мере неожиданно, но первоначальной побудительной причиной создания Java был вовсе не Интернет! Взамен главной движущей силой стала потребность в независимом от платформы (т.е. архитектурно-нейтральном) языке, который можно было бы использовать для построения программного обеспечении, встраиваемого в разнообразные бытовые электронные устройства, такие как микроволновые печи и пульты дистанционного управления. Вероятно, вы уже догадались, что в качестве контроллеров применялись процессоры многих разных типов. Проблема с языками C и C++ (и большин-

ством других языков того времени) заключалась в том, что написанный на них код должен был компилироваться для конкретной целевой платформы. Хотя программу на C++ можно компилировать практически для любого типа процессора, для этого необходим полный компилятор C++, ориентированный на такой тип процессора. Дело в том, что создание компиляторов сопряжено с высокими затратами. Требовалось более простое и менее дорогое решение. В попытке найти подходящее решение Гослинг и другие начали работу над переносимым, независимым от платформы языком, который можно было бы использовать для выпуска кода, способного выполняться под управлением разнообразных процессоров в отличающихся средах. В конечном итоге их усилия привели к созданию Java.

Примерно в то время, когда прорабатывались детали Java, появился второй и, как выяснилось, более важный фактор, который сыграет решающую роль в будущем Java. Второй движущей силой стала, конечно же, Всемирная паутина (она же веб-сеть). Если бы формирование веб-сети не происходило почти одновременно с реализацией Java, то язык Java мог остаться полезным, но малоизвестным языком для программирования бытовой электроники. Однако благодаря появлению веб-сети Java вышел на передний край проектирования языков программирования, т.к. веб-сеть тоже нуждалась в переносимых программах.

Большинство программистов еще в начале своей карьеры осознают, что переносимые программы в равной степени и желательны, и недостижимы. Хотя поиск способа создания эффективных и переносимых (независимых от платформы) программ велся с момента появления самой дисциплины программирования, он отодвигался на второй план другими, более срочными проблемами. Кроме того, поскольку (в то время) почти весь компьютерный мир был разделен на три конкурирующих лагеря, Intel, Macintosh и UNIX, большинство программистов оставались в рамках своих укрепленных границ, что снижало срочность потребности в переносимом коде. Тем не менее, с появлением Интернета и веб-сети старая проблема переносимости возвратилась с новой силой. В конце концов, Интернет представляет собой многообразную распределенную вселенную, наполненную компьютерами, ОС и процессорами различных типов. Несмотря на то что к Интернету подключается много видов платформ, пользователи хотели бы иметь возможность запускать на всех платформах одну и ту же программу. То, что раньше было досадной, но низкоприоритетной проблемой, превратилось в важную необходимость.

К 1993 году членам группы проектировщиков Java стало очевидно, что проблемы переносимости, часто возникающие при написании кода для встраиваемых контроллеров, также обнаруживаются при попытке создания кода для Интернета. По сути, ту же самую проблему, для решения которой изначально проектировался язык Java в мелком масштабе, можно было применить к Интернету в большем масштабе. Это осознание привело к тому, что основное внимание Java переключилось с бытовой электроники на программирование

для Интернета. Таким образом, хотя первоначальной мотивацией было стремление создать архитектурно-нейтральный язык программирования, именно Интернет в конечном итоге привел к крупномасштабному успеху Java.

Как упоминалось ранее, язык Java наследует многие характеристики от языков C и C++. Так было задумано. Проектировщики Java знали, что использование знакомого синтаксиса C и повторение объектно-ориентированных возможностей C++ сделают их язык привлекательным для многочисленных опытных программистов на C/C++. Помимо внешнего сходства языку Java присущи другие особенности, которые обеспечили успех языкам C и C++. Во-первых, Java проектировался, тестировался и совершенствовался практикующими программистами. Он представляет собой язык, основанный на потребностях и опыте людей, которые его придумали. Следовательно, Java — язык для программистов. Во-вторых, Java является целостным и логически непротиворечивым. В-третьих, за исключением ограничений, налагаемых средой Интернета, язык Java предоставляет вам как программисту полный контроль. Если вы программируете хорошо, тогда это отразится на ваших программах. Если вы программируете плохо, то и это скажется на ваших программах. Другими словами, Java — язык не для содействия обучению, а язык для профессиональных программистов.

Из-за сходств языков Java и C++ возникает соблазн считать Java просто "версией C++ для Интернета". Однако думать так было бы крупным заблуждением. В языке Java имеются существенные практические и философские отличия. Хотя язык C++ действительно повлиял на Java, язык Java не является расширенной версией C++. Например, Java не обладает ни прямой, ни обратной совместимостью с C++. Конечно, сходства с C++ значительны, и если у вас есть опыт программирования на C++, тогда с Java вы будете чувствовать себя как дома. И еще один момент: Java не проектировался для замены C++. Язык Java был разработан для решения определенного множества задач. Язык C++ создавался для решения другого множества задач. Оба они будут сосуществовать в течение многих лет.

Как упоминалось в начале главы, языки программирования развиваются по двум причинам: адаптация к изменениям среды и реализация новых идей в области программирования. Изменением среды, которое побудило к созданию Java, была потребность в независимых от платформы программах, предназначенных для распространения в Интернете. Тем не менее, Java также олицетворяет изменения в подходе к написанию программ. Скажем, в языке Java улучшена и усовершенствована объектно-ориентированная парадигма, используемая в C++, добавлена интегрированная поддержка многопоточности и предоставлена библиотека, которая упрощает доступ в Интернет. Однако в конечном итоге не индивидуальные особенности языка Java сделали его настолько замечательным, а скорее весь язык в целом. Java был идеальным ответом на возникшие в то время потребности сильно распределенной компьютерной вселенной. Язык Java стал в программировании для Интернета тем, чем язык C был для системного программирования: революционной силой, которая изменила мир.

Связь с языком C#

Охват и мощь Java продолжили оказывать влияние на мир разработки языков программирования. Многие новаторские возможности, конструкции и концепции Java стали основой для любого нового языка. Просто успех Java слишком важен, чтобы его можно было игнорировать.

Пожалуй, самым значительным примером влияния Java следует считать язык C#. Созданный в Microsoft для поддержки .NET Framework, язык C# тесно связан с Java. Скажем, оба языка имеют тот же общий синтаксис, поддерживают распределенное программирование и задействуют одну и ту же объектную модель. Разумеется, между Java и C# существуют отличия, но в целом "внешний вид и поведение" этих языков очень похожи. Такое "перекрестное опыление" Java и C# является наиболее веским свидетельством того, что язык Java изменил наше представление об использовании языков программирования.

Влияние языка Java на Интернет

Интернет помог языку Java выйти на передний край программирования, а язык Java в свою очередь оказал глубокое влияние на Интернет. Помимо упрощения программирования для веб-сети в целом благодаря Java появился новый вид сетевых программ, названных аплетами, которые изменили представление онлайнового мира о содержимом. Кроме того, язык Java решил ряд самых трудных проблем, связанных с Интернетом: переносимость и безопасность. Давайте рассмотрим каждую проблему по отдельности.

Аплеты Java

На момент создания Java одной из наиболее захватывающих возможностей был аплет. Аплет — это особый вид программы на Java, который предназначен для передачи через Интернет и автоматического выполнения внутри веб-браузера, совместимого с Java. После щелчка пользователем на ссылке, содержащей аплет, он загружается и запускается в браузере. Аплеты задумывались как небольшие программы. Они обычно применяются для отображения данных, предоставленных сервером, обработки пользовательского ввода или поддержки простых функций, таких как расчет процента по кредиту, которые выполняются локально, а не на сервере. По существу аплет позволяет переместить определенную функциональность из серверной стороны на клиентскую сторону.

Появление аплетов было важным, потому что в то время они расширяли совокупность объектов, которые можно было свободно перемещать в киберпространстве. Вообще говоря, есть две очень крупные категории объектов, передаваемых между сервером и клиентом: пассивная информация и динамические, активные программы. Например, когда вы читаете сообщения электронной почты, то просматриваете пассивные данные. Даже при загрузке программы ее код по-прежнему будет только пассивными данными, пока вы

не запустите программу. Напротив, аплет представляет собой динамическую самостоятельно запускающуюся программу. Такая программа является активным агентом на клиентском компьютере, но инициируется сервером.

В самом начале существования Java аплеты были важной частью программирования на Java. Они продемонстрировали мощь и преимущества Java, увеличили привлекательность веб-страниц и позволили программистам в полной мере изучить возможности Java. Несмотря на то что аплеты все еще используются в наши дни, с течением времени они стали менее важными. По причинам, которые будут объясняться позже, в версии JDK 9 началась фаза постепенного вывода аплетов из употребления, а в версии JDK 11 поддержка аплетов была прекращена.

Безопасность

Какими бы желанными ни были динамические сетевые программы, они также могут порождать серьезные проблемы в областях безопасности и переносимости. Вполне очевидно, что необходимо не допустить нанесение ущерба программой, которая загружается и выполняется на клиентском компьютере. Кроме того, программа должна иметь возможность запускаться в разнообразных средах и под управлением разных ОС. Вы увидите, что Java эффективно и элегантно решает эти задачи. Взглянем на них более пристально, начав с безопасности.

Наверняка вам уже известно, что каждый раз, когда вы загружаете с виду "нормальную" программу, то идете на риск, поскольку загружаемый код может содержать вирус, "троянский конь" или другой вредоносный код. В основе проблемы лежит тот факт, что вредоносный код способен нанести ущерб, получив несанкционированный доступ к системным ресурсам. Скажем, вирусная программа может собирать конфиденциальную информацию вроде номеров кредитных карт, остатков на банковских счетах и паролей, проводя поиск в локальной файловой системе вашего компьютера. Чтобы обеспечить безопасную загрузку и выполнение программ Java на клиентском компьютере, нужно было предотвратить запуск атак подобного рода.

Такая защита достигается в Java за счет того, что вы можете ограничивать функционирование приложения средой выполнения Java и запрещать ему доступ к другим областям компьютера. (Вскоре вы узнаете, как это делать.) Возможность загрузки программ с определенной уверенностью в том, что они не нанесут никакого вреда, вероятно, считается наиболее новаторским аспектом Java.

Переносимость

Переносимость — важный аспект Интернета, т.к. к нему подключается много разнотипных компьютеров и ОС. Если программа на Java должна запускаться практически на любом компьютере, подключенном к Интернету, тогда должен быть какой-то способ разрешать ее выполнение в разных систе-

мах. Другими словами, необходим механизм, который позволит одному приложению загружаться и выполняться разнообразными процессорами, ОС и браузерами. Нецелесообразно иметь разные версии приложения для разных компьютеров. Один и тот же прикладной код обязан работать на всех компьютерах. Следовательно, потребовались средства генерации переносимого исполняемого кода. Как вы вскоре увидите, тот же самый механизм, который помогает обеспечивать безопасность, содействует и переносимости.

Магия Java: байт-код

Основная особенность Java, которая позволила решить только что описанные проблемы переносимости и безопасности, заключается в том, что компилятор Java генерирует не исполняемый код, а байт-код. Байт-код представляет собой оптимизированный набор инструкций, предназначенных для выполнения так называемой виртуальной машиной Java (Java Virtual Machine — JVM), которая является частью исполняющей среды Java (Java Runtime Environment — JRE). По существу исходная версия JVM проектировалась как интерпретатор для байт-кода. Это может несколько удивить, поскольку многие современные языки предусматривают компиляцию в исполняемый код из соображений безопасности. Тем не менее, тот факт, что программа на Java выполняется машиной JVM, помогает решить основные проблемы, связанные с программами для веб-сети. Рассмотрим причины.

Трансляция программы на Java в байт-код значительно упрощает ее запуск в разнообразных средах, потому что для каждой платформы необходимо реализовать только машину JVM. Когда для заданной системы существует среда JRE, под ее управлением можно запускать любую программу на Java. Однако, хотя детали реализации машин JVM будут отличаться от платформы к платформе, все они воспринимают один и тот же байт-код Java. Если бы программа на Java компилировалась в машинный код, тогда для каждого типа процессора, подключенного к Интернету, должна была существовать своя версия программы. Конечно же, это нельзя считать приемлемым решением. Таким образом, выполнение байт-кода машиной JVM является самым легким способом получения по-настоящему переносимых программ.

Факт выполнения программы на Java машиной JVM также делает ее безопасной. Поскольку машина JVM находится под контролем, она управляет выполнением программы. В итоге у JVM появляется возможность создать ограниченную среду выполнения, называемую песочницей, которая содержит программу, препятствуя неограниченному доступу к машине. Кроме того, повышается и безопасность за счет определенных ограничений, имеющихся в языке Java.

Вообще говоря, когда программа компилируется в промежуточную форму и затем интерпретируется виртуальной машиной, она выполняется медленнее, чем в ситуации, если бы она компилировалась в исполняемый код. Тем не менее, в случае Java разница не настолько велика. Из-за того, что байт-код в

высшей степени оптимизирован, его применение позволяет машине JVM выполнять программы гораздо быстрее, нежели вы могли бы ожидать.

Хотя язык Java был спроектирован как интерпретируемый, нет ничего, что помешало бы компилировать байт-код на лету в машинный код с целью повышения производительности. По этой причине вскоре после первоначального выпуска Java была представлена технология HotSpot, которая предлагала оперативный (Just-In-Time — JIT) компилятор для байт-кода. Когда компилятор JIT входит в состав машины JVM, избранные порции байт-кода компилируются в исполняемый код в режиме реального времени, часть за частью по запросу. Важно понимать, что программа на Java не компилируется в исполняемый код сразу целиком. Взамен компилятор JIT компилирует код по мере необходимости во время выполнения. Более того, компилируются не все последовательности байт-кода, а только те, которые извлекут пользу из компиляции. Остальной код просто интерпретируется. Однако подход JIT все же обеспечивает значительный рост производительности. Даже при динамической компиляции байт-кода характеристики переносимости и безопасности сохраняются, т.к. машина JVM по-прежнему несет ответственность за среду выполнения.

Еще один момент: проводились эксперименты с ранней (ahead-of-time) компиляцией для Java. Такой компилятор можно использовать для компиляции байт-кода в машинный код перед выполнением машиной JVM, а не на лету. Некоторые предшествующие версии JDK поставлялись с экспериментальным ранним компилятором; тем не менее, в версии JDK 17 он был удален. Ранняя компиляция — это специализированная возможность, и она не заменяет описанный выше традиционный подход Java. По причине узкоспециализированной природы ранняя компиляция в книге обсуждаться не будет.

Выход за рамки аплетов

На момент написания этой книги с момента исходного выпуска Java прошло более двух десятилетий. За прошедшие годы произошло много изменений. Во времена создания Java Интернет был захватывающей инновацией, веб-браузеры быстро развивались и совершенствовались, современная форма смартфона еще не была изобретена, а до повсеместного применения компьютеров оставалось еще несколько лет. Как и следовало ожидать, язык Java тоже изменился вместе со способом его использования. Возможно, ничто не иллюстрирует текущую эволюцию лучше, чем аплет.

Как объяснялось ранее, в начальные годы существования Java аплеты были важной составляющей программирования на Java. Они не только увеличивали привлекательность веб-страниц, но стали крайне заметной особенностью языка Java, повышая его популярность. Однако аплеты полагаются на подключаемый модуль браузера для Java. Таким образом, чтобы аплет работал, браузер должен его поддерживать. В течение последних нескольких лет поддержка подключаемого модуля браузера для Java ослабла. Проще говоря,

без поддержки браузера аплеты нежизнеспособны. По этой причине в версии JDK 9 начался поэтапный отказ от аплетов, и поддержка аплетов была объявлена нерекомендуемой. В языке Java нерекомендуемое средство означает, что оно все еще доступно, но помечено как устаревшее. Следовательно, в новом коде нерекомендуемое средство применяться не должно. Поэтапный отказ завершился с выходом JDK 11, поскольку поддержка аплетов исполняющей средой была удалена. Начиная с версии JDK 17, весь API-интерфейс аплетов (Applet API) стал нерекомендуемым и подлежащим удалению.

Интересно отметить, что спустя несколько лет после создания в Java была добавлена альтернатива аплетам, которая называлась Java Web Start и позволяла динамически загружать приложение из веб-страницы. Она представляла собой механизм развертывания, особенно удобный в случае крупных приложений Java, которые не подходили для реализации в виде аплетов. Разница между аплетом и приложением Web Start заключалась в том, что приложение Web Start выполнялось само по себе, а не внутри браузера. Таким образом, оно выглядело во многом похоже на "нормальное" приложение. Тем не менее, это требовало наличия в размещающей системе автономной среды JRE, поддерживающей Web Start. В версии JDK 11 поддержка Java Web Start была удалена.

Учитывая, что современные версии Java не поддерживают ни аплеты, ни Java Web Start, вас может интересовать, какой механизм должен использоваться для развертывания приложения Java. На момент написания книги частью ответа был инструмент `jlink`, добавленный в версии JDK 9. С его помощью можно создавать полный образ, который включает всю необходимую поддержку для вашей программы, в том числе среду JRE. Другая часть ответа — инструмент `jpackage`, появившийся в версии JDK 16. Его можно применять для создания готового к установке приложения. Хотя подробное обсуждение стратегий развертывания выходит за рамки настоящей книги, в будущем вам придется уделить внимание данной теме.

Более быстрый график выпуска

Недавно в Java произошло еще одно крупное изменение, но оно не касается языка или исполняющей среды. Взамен изменение связано со способом планирования выпусков Java. В прошлом между основными выпусками Java обычно проходило два и больше лет. Однако после выхода JDK 9 промежутки времени между основными выпусками Java сократились. В наши дни ожидается, что основной выпуск будет происходить по строгому графику, и расчетное время между выпусками составит всего лишь полгода.

Каждый полугодичный выпуск, теперь называемый выпуском функциональных средств, будет включать те средства, которые готовы к моменту выпуска. Такая увеличенная частота выпусков позволит программистам на Java получать своевременный доступ к новым средствам и улучшениям. Кроме того, у Java появится возможность быстро реагировать на запросы постоянно меняющейся программной среды. Выражаясь просто, более быстрый график выпуска обещает стать очень позитивным событием для программистов на Java.

Ожидается, что каждые три года будет производиться выпуск с долгосрочной поддержкой (long-term support — LTS). Выпуск LTS будет поддерживаться (и, следовательно, оставаться жизнеспособным) в течение периода времени, превышающего полгода. Первым выпуском LTS был JDK 11. Вторым выпуском LTS стал JDK 17, с учетом которого была обновлена эта книга. Из-за стабильности, предлагаемой выпуском LTS, вполне вероятно, что его набор средств будет определять базовый уровень функциональности для промежутка в несколько лет. Последние сведения о долгосрочной поддержке и графике выхода выпусков LTS ищите на веб-сайте Oracle.

В текущее время выпуски функциональных средств запланированы на март и сентябрь каждого года. В результате JDK 10 вышел в марте 2018 года, т.е. через полгода после выхода JDK 9. Следующий выпуск (JDK 11) вышел в сентябре 2018 года и был выпуском LTS. Затем последовал JDK 12 в марте 2019 года, JDK 13 в сентябре 2019 года и т.д. На момент написания книги последним выпуском был JDK 17, который является выпуском LTS. Опять-таки ожидается, что каждые полгода будет выходить новый выпуск функциональных средств. Конечно, имеет смысл ознакомиться с актуальной информацией о графике выпусков.

Во время написания книги на горизонте показалось несколько новых функциональных средств Java. По причине более быстрого графика выпусков с высокой вероятностью можно предположить, что некоторые из них будут добавлены в Java в течение ближайших нескольких лет. Рекомендуется внимательно просматривать сведения и замечания по каждому полугодовому выпуску. Сейчас действительно самое подходящее время, чтобы стать программистом на Java!

Сервлеты: Java на серверной стороне

Код на клиентской стороне — лишь одна половина уравнения "клиент-сервер". Довольно скоро после первоначального выпуска Java стало очевидно, что язык Java будет полезен и на серверной стороне. Одним из результатов стал сервлет, представляющий собой небольшую программу, которая выполняется на сервере.

Сервлеты используются для создания динамически генерируемого содержимого, впоследствии передаваемого клиенту. Например, онлайновый магазин может применять сервлет для поиска цены товара в базе данных. Затем информация о цене используется для динамического генерирования веб-страницы, которая отправляется браузеру. Несмотря на то что динамически генерируемое содержимое было доступно через механизмы вроде CGI (Common Gateway Interface — интерфейс общего шлюза), с сервлетом связано несколько преимуществ, включая увеличенную производительность.

Поскольку сервлеты (подобно всем программам на Java) компилируются в байт-код и выполняются машиной JVM, они обладают высокой переносимостью. Таким образом, один и тот же сервлет может применяться в разно-

образных серверных средах. Единственное требование — поддержка на сервере машины JVM и контейнера сервлетов. В настоящее время код серверной стороны в целом является основным использованием Java.

Терминология языка Java

Никакое обсуждение истории Java не будет полным без рассмотрения терминологии Java. Хотя основными факторами, вызвавшими изобретение Java, были переносимость и безопасность, важную роль в формировании финальной формы языки сыграли и другие факторы. Команда разработчиков Java подытожила ключевые соображения в виде следующего списка терминов:

- простота;
- безопасность;
- переносимость;
- объектная ориентация;
- надежность;
- многопоточность;
- нейтральность к архитектуре;
- интерпретируемость;
- высокая производительность;
- распределенность;
- динамичность.

Два термина из списка уже обсуждались: безопасность и переносимость. Давайте выясним, что подразумевается под остальными.

Простота

Язык Java был спроектирован так, чтобы быть легким в изучении и эффективным в использовании профессиональным программистом. Если у вас есть определенный опыт программирования, то освоить Java не составит особого труда. Если вы уже понимаете базовые концепции ООП, тогда изучение Java пройдет еще проще. Лучше всего то, что если вы — опытный программист на C++, то переход на Java потребует совсем небольших усилий. Поскольку язык Java унаследовал синтаксис C/C++ и многие объектно-ориентированные возможности C++, у большинства программистов не будет возникать проблем с изучением Java.

Объектная ориентация

Несмотря на влияние своих предшественников, язык Java не проектировался так, чтобы быть совместимым на уровне исходного кода с любым дру-

гим языком. Это дало команде разработчиков Java свободу проектирования с чистого листа. Результатом стал ясный, удобный и прагматичный подход к объектам. Обильно заимствуя из многих продуктивных объектно-ориентированных сред, существующих на протяжении последних нескольких десятилетий, в Java удалось найти баланс между парадигмой "все является объектом" сторонников чистоты стиля и более прагматичной моделью "не путайтесь под ногами". Объектная модель в Java проста и легко расширяема, в то время как элементарные типы, такие как целочисленные, были сохранены высокопроизводительными сущностями, которые не являются объектами.

Надежность

Многоплатформенная среда веб-сети предъявляет к программе необычные требования, потому что программа должна надежно выполняться в разнообразных системах. Таким образом, при проектировании Java возможности создания надежных программ был назначен высокий приоритет. Для обеспечения надежности Java ограничивает вас в ряде ключевых областей, чтобы вынудить искать свои просчеты на ранней стадии проектирования программ. В то же время Java избавляет вас от необходимости беспокоиться о многих наиболее распространенных причинах ошибок при программировании. Поскольку Java — строго типизированный язык, ваш код проверяется на этапе компиляции. Тем не менее, код также проверяется и во время выполнения. Многие трудно обнаруживаемые ошибки, которые часто приводят к возникновению сложных для воспроизведения ситуаций во время выполнения, в Java попросту невозможны. Предсказуемость поведения написанного вами кода в несходных условиях — ключевая особенность Java.

Чтобы лучше понять, как обеспечивается надежность в Java, рассмотрим две главных причины отказа программ: просчеты в управлении памятью и неправильно обработанные исключительные ситуации (т.е. ошибки времени выполнения). В традиционных программных средах управление памятью может оказаться сложной и утомительной задачей. Скажем, в C/C++ программист будет часто вручную выделять и освобождать динамическую память. Подход подобного рода иногда приводит к возникновению проблем, потому что программисты будут либо забывать об освобождении ранее выделенной памяти, либо, что хуже, пытаться освободить память, которая все еще задействована в другой части кода. Java практически устраняет указанные проблемы, самостоятельно управляя выделением и освобождением памяти. (На самом деле освобождение выполняется полностью автоматически, поскольку Java обеспечивает сборку мусора для неиспользуемых объектов.) Условия для исключений в традиционных средах часто возникают в ситуациях вроде деления на ноль или отсутствия нужного файла, и справляться с ними приходится с помощью неуклюжих и трудных для чтения конструкций. Java помогает и этой области, предлагая объектно-ориентированную обработку исключений. В хорошо написанной программе на Java все ошибки времени выполнения могут — и должны — обрабатываться вашей программой.

Многопоточность

Язык Java был спроектирован с целью удовлетворения реальной потребности в создании интерактивных сетевых программ. Для достижения такой цели в Java поддерживается многопоточное программирование, которое дает возможность писать программы, выполняющие много действий одновременно. Исполняющая среда Java поддерживает элегантное, но вместе с тем сложное решение для синхронизации множества процессов, которое делает возможным построение бесперебойно работающих интерактивных систем. Простой в применении подход к многопоточности в Java позволяет вам сосредоточить внимание на конкретном поведении программы, а не думать о многозадачной подсистеме.

Нейтральность к архитектуре

Центральной задачей проектировщиков Java было обеспечение долговечности и переносимости кода. На момент создания Java одной из главных проблем, стоявших перед программистами, было отсутствие всяких гарантий того, что программа, написанная сегодня, будет выполняться завтра — даже на той же самой машине. Обновления ОС, модернизация процессоров и изменения в ключевых системных ресурсах могут в совокупности привести к нарушению работоспособности программы. Пытаясь изменить сложившуюся в то время ситуацию, проектировщикам пришлось принять ряд жестких решений в отношении языка Java и машины JVM. Они преследовали цель "написанное однажды выполняется везде, в любое время, всегда". В значительной степени эта цель была достигнута.

Интерпретируемость и высокая производительность

Как было показано ранее, язык Java делает возможным создание межплатформенных программ за счет их компиляции в промежуточное представление, называемое байт-кодом Java. Байт-код может выполняться на любой системе, где внедрена машина JVM. В большинстве попыток построения межплатформенных решений это делалось ценой снижения производительности. Как объяснялось в начале главы, байт-код Java был тщательно спроектирован, чтобы легко транслироваться прямо в машинный код для достижения очень высокой производительности с использованием оперативного компилятора. Исполняющие среды Java, предлагающие такую возможность, не утрачивают преимуществ независимого от платформы кода.

Распределенность

Язык Java проектировался для распределенной среды Интернета, поскольку он поддерживает протоколы TCP/IP. Фактически доступ к ресурсу с применением URL мало чем отличается от доступ к файлу. В Java также поддерживается средство удаленного вызова методов (Remote Method Invocation — RMI), которое позволяет программе вызывать методы через сеть.

Динамичность

Программы на Java содержат существенный объем информации о типах времени выполнения, используемой для проверки и разрешения доступа к объектам во время выполнения, что делает возможным безопасное и надлежащее динамическое связывание кода. Это критически важно для надежности среды Java, в которой небольшие фрагменты байт-кода могут динамически обновляться в функционирующей системе.

Эволюция языка Java

Первоначальный выпуск Java по праву считался революционным, но он не ознаменовал собой конец эпохи быстрых инноваций Java. В отличие от большинства других программных систем, в которых обычно происходили небольшие поэтапные улучшения, язык Java продолжил развиваться взрывными темпами. Вскоре после выпуска Java 1.0 проектировщики уже создали Java 1.1. Возможности, добавленные в Java 1.1, оказались более значительными и существенными, чем можно было судить по увеличению младшего номера версии. В Java 1.1 было добавлено много новых библиотечных элементов, изменен способ обработки событий и переконфигурированы многочисленные средства из библиотеки Java 1.0. Кроме того, несколько средств, определенных в Java 1.0, стали нерекомендуемыми (признаны устаревшими). Таким образом, в Java 1.1 одновременно были добавлены и удалены характеристики исходной спецификации.

Следующим крупным выпуском Java стал Java 2, где "2" означает "второе поколение". Создание Java 2 стало переломным событием, положившим начало "современной эпохи" Java. Первый выпуск Java 2 имел номер версии 1.2. Может показаться странным, что для первого выпуска Java 2 использовался номер версии 1.2. Причина в том, что первоначально он относился к внутреннему номеру версии библиотек Java, но потом был обобщен для ссылки на целый выпуск. С выходом Java 2 в компании Sun перепаковали продукт Java как J2SE (Java 2 Platform Standard Edition — платформа Java 2, стандартная редакция) и номера версий стали применяться к данному продукту.

В Java 2 была добавлена поддержка нескольких новых средств, включая Swing и Collections Framework, а также усовершенствована машина JVM и различные программные инструменты. Кроме того, в Java 2 несколько средств стали нерекомендуемыми. Наиболее важные изменения коснулись класса `Thread`, в котором методы `suspend()`, `resume()` и `stop()` были объявлены нерекомендуемыми.

Первым крупным обновлением первоначального выпуска Java 2 стала версия J2SE 1.3. По большей части она расширяла существующую функциональность и "усиливала" среду разработки. В общем случае программы, написанные для версий 1.2 и 1.3, совместимы по исходному коду. Хотя версия 1.3 содержала меньший набор изменений, чем предшествующие три крупных выпуска, она все-таки была важна.

Выпуск J2SE 1.4 дополнительно улучшил Java. Он содержал несколько важных обновлений, улучшений и дополнений. Например, в J2SE 1.4 было введено ключевое слово `assert`, цепочки исключений и подсистема ввода-вывода на основе каналов. Кроме того, были внесены изменения в Collections Framework и в классы для работы с сетью. Вдобавок повсюду встречаются многочисленные мелкие изменения. Несмотря на значительное количество новых средств, версия 1.4 сохранила почти стопроцентную совместимость по исходному коду с предшествующими версиями.

Следующим выпуском был J2SE 5, ставший революционным. В отличие от большинства предшествующих обновлений Java, которые предлагали важные, но умеренные улучшения, выпуск J2SE 5 фундаментальным образом расширил границы, возможности и область использования языка. Чтобы оценить значимость изменений, которые были внесены в Java в выпуске J2SE 5, взгляните на следующий список, где перечислены только основные из всех новых функциональных средств:

- обобщения;
- аннотации;
- автоупаковка и автораспаковка;
- перечисления;
- расширенный цикл `for` в стиле "for-each";
- аргументы переменной длины (vararg);
- статический импорт;
- форматированный ввод-вывод;
- утилиты параллелизма.

Мелкие подстройки или поэтапные обновления в списке не указаны. Каждый элемент списка представляет значительное дополнение языка Java. Некоторые из них, такие как обобщения, расширенный цикл `for` и аргументы переменной длины, вводили новые синтаксические элементы. Другие, в числе которых автоупаковка и автораспаковка, изменяли семантику языка. Аннотации привнесли в программирование совершенно новое измерение. Во всех случаях влияние этих дополнений вышло за рамки их прямого воздействия. Они изменили сам характер Java.

Важность упомянутых выше новых средств отражена в назначенном номере версии — 5. В других условиях следующим номером версии Java был бы 1.5. Однако новые средства были настолько значительными, что переход от версии 1.4 к 1.5, казалось, просто не смог бы выразить масштаб изменений. Взамен в Sun предпочли увеличить номер версии до 5, чтобы подчеркнуть тот факт, что произошло важное событие. Таким образом, выпуск получил название J2SE 5, а комплект разработчика — JDK 5. Тем не менее, ради соблюдения согласованности в Sun решили использовать 1.5 в качестве внутреннего номера версии, который также называют номером версии разработчиков. Цифра 5 в J2SE 5 называется номером версии продукта.

Следующий выпуск Java получил имя Java SE 6. В Sun снова решили изменить название платформы Java. Прежде всего, обратите внимание, что цифра 2 была отброшена. Соответственно, платформа стала называться Java SE, а официальный продукт — Java Platform, Standard Edition 6 (платформа Java, стандартная редакция). Комплекту разработчика на Java (Java Development Kit) было дано название JDK 6. По аналогии с J2SE 5 цифра 6 в Java SE 6 — это номер версии продукта. Внутренний номер версии разработчиков — 1.6.

Выпуск Java SE 6 построен на основе J2SE 5 с добавлением поэтапных усовершенствований. Он не дополнял какими-либо важными средствами сам язык Java, но расширил библиотеки API, добавил ряд новых пакетов и предоставил усовершенствования исполняющей среды. Кроме того, в течение своего длинного (в понятиях Java) жизненного цикла он прошел через несколько модернизаций, попутно добавив некоторое количество обновлений. В целом выпуск Java SE 6 послужил дальнейшему укреплению достижений выпуска J2SE 5.

Очередным выпуском Java стал Java SE 7 с названием комплекта разработчика JDK 7 и внутренним номером версии 1.7. Выпуск Java SE 7 был первым крупным выпуском Java после того, как компанию Sun Microsystems приобрела компания Oracle. Выпуск Java SE 7 содержал много новых функциональных средств, включая существенные дополнения языка и библиотек API. Кроме того, в состав Java SE 7 входили обновления исполняющей среды Java, которые обеспечивали поддержку языков, отличающихся от Java, но наибольший интерес у программистов на Java вызывали языковые и библиотечные дополнения.

Новые языковые средства были разработаны в рамках проекта Project Coin. Целью проекта Project Coin была идентификация ряда небольших изменений языка Java, подлежащих включению в JDK 7. Хотя в совокупности эти изменения называли "небольшими", эффект от них был довольно значительным с точки зрения кода, на который они влияли. Фактически для многих программистов такие изменения вполне могли стать самыми важными новыми средствами в Java SE 7. Ниже приведен список языковых средств, добавленных комплектом JDK 7.

- Наделение типа `String` способностью управлять оператором `switch`.
- Двоичные целочисленные литералы.
- Символы подчеркивания в числовых литералах.
- Расширение оператора `try` под название `try` с ресурсами, которое поддерживает автоматическое управление ресурсами. (Например, потоки могут автоматически закрываться, когда они больше не нужны.)
- Выведение типов (через ромбовидную операцию) при конструировании экземпляра обобщенного типа.
- Усовершенствованная обработка исключений, при которой два или большее количество исключений можно перехватывать одним оператором `catch` (многократный перехват), и улучшенная проверка типов для исключений, генерируемых повторно.

- Хотя это не является изменением синтаксиса, была улучшена информативность выдаваемых компилятором предупреждений, связанных с некоторыми типами методов с аргументами переменной длины, и вы имеете больший контроль над предупреждениями.

Как видите, несмотря на то, что средства Project Coin считались небольшими изменениями, внесенными в язык, их преимущества были гораздо значительнее, чем можно было бы предположить по характеристике "небольшие". В частности, оператор `try` с ресурсами основательно повлиял на способ написания кода, базирующегося на потоках. Кроме того, возможность использования типа `String` для управления оператором `switch` оказалась долгожданным усовершенствованием, которое во многих ситуациях упрощало написание кода.

Выпуск Java SE 7 содержал несколько дополнений библиотеки Java API. Двумя наиболее важными из них были усовершенствования инфраструктуры NIO Framework и дополнение инфраструктуры Fork/Join Framework. Инфраструктура NIO Framework (аббревиатура NIO первоначально означала New I/O (новый ввод-вывод)) была добавлена в версии Java 1.4. Однако изменения, внесенные выпуском Java SE 7, в корне расширили ее возможности. Изменения были настолько значительными, что часто применяется термин NIO.2.

Инфраструктура Fork/Join Framework обеспечивает важную поддержку параллельного программирования. Параллельное программирование — это название, обычно относящееся к методикам, которые позволяют эффективно эксплуатировать компьютеры, содержащие более одного процессора, в том числе многоядерные системы. Преимущество многоядерных сред заключается в возможности значительного повышения производительности программ. Инфраструктура Fork/Join Framework содействует параллельному программированию за счет того, что:

- упрощает создание и использование задач, которые могут выполняться параллельно;
- автоматически задействует множество процессоров.

Следовательно, с применением Fork/Join Framework вы можете легко создавать масштабируемые приложения, которые автоматически извлекают выгоду из процессоров, доступных в среде выполнения. Конечно, не все алгоритмы поддаются распараллеливанию, но для тех, которые это допускают, можно добиться существенного улучшения в скорости выполнения.

Следующим выпуском Java был Java SE 8 с комплектом разработчика JDK 8. Он имел внутренний номер версии 1.8. Комплект JDK 8 стал значительным обновлением языка Java из-за включения многообещающего нового языкового средства: лямбда-выражений. Влияние лямбда-выражений было и продолжило быть весьма существенным, изменяя как способ концептуализации программных решений, так и способ написания кода Java. В главе 15 будет показано, что лямбда-выражения добавляют в язык Java возможности функ-

ционального программирования. Помимо прочего, лямбда-выражения содействуют упрощению и сокращению объема исходного кода, необходимого для создания определенных конструкций, таких как некоторые виды анонимных классов. Добавление лямбда-выражений также привело к появлению в языке новой операции (->) и нового элемента синтаксиса.

Появление лямбда-выражений оказало широкомасштабное влияние и на библиотеки Java, в которые были добавлены новые средства, выгодно использующие лямбда-выражения. Одним из самых важных средств считается новый потоковый API, находящийся в пакете `java.util.stream`. Потоковый API поддерживает конвейерные операции с данными и оптимизирован под лямбда-выражения. В еще одном новом пакете `java.util.function` определено несколько функциональных интерфейсов, которые предоставляют дополнительную поддержку для лямбда-выражений. В библиотеке API можно обнаружить и другие новые средства, связанные с лямбда-выражениями.

Еще одна особенность, обусловленная добавлением лямбда-выражений, касается интерфейса. Начиная с JDK 8, появилась возможность определять стандартную реализацию для метода, объявленного в интерфейсе. Если реализация такого метода в классе не создавалась, тогда используется стандартная реализация из интерфейса. Данное средство позволяет элегантно развивать интерфейс с течением времени, поскольку в интерфейс можно добавить новый метод, не нарушив работу существующего кода. Оно также упрощает реализацию интерфейса в ситуации, когда стандартные методы подходят. Другими новыми средствами в JDK 8 были, помимо прочих, новый API для даты и времени, аннотации типов и возможность организации параллельной обработки при сортировке массива.

Следующим выпуском Java стал Java SE 9. Комплект разработчика назывался JDK 9, а внутренним номером версии также был 9. Комплект JDK 9 представлял крупный выпуск Java, включая значительные улучшения как языка Java, так и его библиотек. Подобно JDK 5 и JDK 8 выпуск JDK 9 коренным образом повлиял на язык Java и его библиотеки.

Главным новым средством JDK 9 были модули, которые позволили указывать взаимосвязи и зависимости кода, составляющего приложение. Модули также добавляют еще одно измерение к средствам контроля доступа Java. Включение модулей привело к добавлению в Java нового элемента синтаксиса и нескольких ключевых слов. Кроме того, в JDK появился инструмент под названием `jlink`, который позволяет программисту создавать для приложения образ времени выполнения, содержащий только необходимые модули. Был введен новый файловый тип JMOD. Модули оказали основательное влияние и на библиотеку API, т.к. начиная с JDK 9, библиотечные пакеты организованы в виде модулей.

Хотя модули являются значительным усовершенствованием Java, они концептуально просты и прямолинейны. Кроме того, поскольку унаследованный код, написанный до появления модулей, полностью поддерживается, модули могут быть интегрированы в процесс разработки постепенно. Нет никакой

необходимости тотчас же изменять любой имеющийся код, чтобы задействовать модули. Короче говоря, модули добавляют важную функциональность, не меняя сущности Java.

Помимо модулей в JDK 9 включено много других новых средств. Особый интерес представляет JShell — инструмент, который поддерживает интерактивное экспериментирование с программами и изучение Java. (Введение в JShell приведено в приложении Б.) Еще одним интересным обновлением является поддержка закрытых методов интерфейсов. Их добавление в дальнейшем расширяет поддержку JDK 8 стандартных методов в интерфейсах. В JDK 9 инструмент `javadoc` был снабжен возможностью поиска, для поддержки которой предусмотрен новый дескриптор под названием `@index`. Как и предшествующие выпуски, JDK 9 вносит несколько усовершенствований в библиотеки Java API.

Обычно в любом выпуске Java наибольший интерес вызывают новые средства. Тем не менее, выпуск JDK 9 примечателен тем, что в нем объявлен устаревшим один заметный аспект Java: аплеты. Начиная с JDK 9, применять аплеты в новых проектах больше не рекомендуется. Как объяснялось ранее в главе, из-за ослабления поддержки аплетов со стороны браузеров (и других факторов) в выпуске JDK 9 объявлен устаревшим весь API для аплетов.

Следующим выпуском Java был Java SE 10 (JDK 10). Ранее уже упоминалось, что начиная с JDK 10, выпуски Java должны выходить по строгому графику с расчетным временем между выпусками, составляющим всего лишь полгода. В результате выпуск JDK 10 появился в марте 2018 года, т.е. спустя полгода после выпуска JDK 9. Главным новым языковым средством, добавленным JDK 10, стала поддержка выведения типов локальных переменных, благодаря которому теперь можно позволить выводить тип локальной переменной из типа ее инициализатора, а не указывать его явно. Для обеспечения такой возможности в Java было добавлено контекстно-чувствительное ключевое слово `var`. Выведение типов способно упростить код за счет устранения необходимости избыточно указывать тип переменной, когда его можно вывести из инициализатора переменной.

Выведение типов также может упростить объявления в ситуациях, при которых тип трудно понять или невозможно указать явно. Выведение типов локальных переменных стало обычной частью современной программной среды. Его включение в Java помогло сохранять Java в актуальном состоянии с учетом меняющихся тенденций в проектировании языка. Наряду с другими изменениями в JDK 10 переопределена строка версии Java с изменением смысла номеров версий, чтобы они лучше соотносились с новым графиком выпуска.

Следующим выпуском Java был Java SE 11 (JDK 11). Он вышел в сентябре 2018 года, т.е. через полгода после JDK 10, и был выпуском LTS. Главным новым языковым средством в JDK 11 являлась поддержка использования ключевого слова `var` в лямбда-выражениях. Вместе с рядом подстроек и обновлений API в целом выпуск JDK 11 добавил новый API для работы с сетью,

который будет интересен широкому кругу разработчиков. Он называется HTTP Client API, находится в пакете `java.net.http` и предоставляет улучшенную, обновленную и усовершенствованную поддержку работы с сетью для клиентов HTTP. Кроме того, загрузчик приложений Java получил еще один режим выполнения, позволяющий напрямую запускать простые однофайловые программы. В JDK 11 также были удалены некоторые средства. Возможно, наибольший интерес представляет удаление поддержки аплетов ввиду его исторической значимости. Вспомните, что аплеты впервые были объявлены нерекомендуемыми в JDK 9. В выпуске JDK 11 поддержка аплетов была удалена. Поддержка другой технологии развертывания, называемой Java Web Start, тоже была удалена в JDK 11. Поскольку среда выполнения продолжила развиваться, и аплеты, и Java Web Start быстро утратили актуальность. Еще одним ключевым изменением в JDK 11 стало то, что JavaFX больше не входит в состав JDK. Взамен эта инфраструктура для построения графических пользовательских интерфейсов превратилась в проект с открытым кодом. Из-за того, что упомянутые средства перестали быть частью JDK, в книге они не обсуждаются.

Между LTS-выпуском JDK 11 и следующим LTS-выпуском (JDK 17) вышли пять выпусков функциональных средств: с JDK 12 до JDK 16. В выпусках JDK 12 и JDK 13 новые языковые средства не добавлялись. В выпуске JDK 14 была добавлена поддержка выражения `switch`, которое представляет собой конструкцию `switch`, производящую значение. Также были проведены другие усовершенствования `switch`. В выпуске JDK 15 появились текстовые блоки, по существу являющиеся строковыми литералами, которые могут занимать более одной строчки. В выпуске JDK 16 операция `instanceof` была расширена сопоставлением с образцом и добавлен новый тип класса под названием запись вместе с новым контекстно-чувствительным ключевым словом `record`. Запись предлагает удобный способ объединения данных. В выпуске JDK 16 также предоставляется новый инструмент пакетирования приложений, называемый `jpackage`.

На момент написания книги самой последней версией Java была Java SE 17 (JDK 17). Как упоминалось ранее, это второй LTS-выпуск Java, что имеет особое значение. Его главное новое средство — возможность запечатывать классы и интерфейсы. Запечатывание дает вам контроль над наследованием класса, а также над наследованием и реализацией интерфейса. Для такой цели были добавлены контекстно-чувствительные ключевые слова `sealed`, `permits` и `non-sealed` (первое ключевое слово Java с дефисом). Кроме того, в JDK 17 произошла пометка API для аплетов как устаревшего и подлежащего удалению. Вы уже знаете, что поддержку аплетов убрали несколько лет назад. Однако API для аплетов просто был помечен как нерекомендуемый, что позволяло компилировать практически исчезающий код, который полагался на этот API. После выхода JDK 17 теперь API для аплетов подлежит удалению в будущем выпуске.

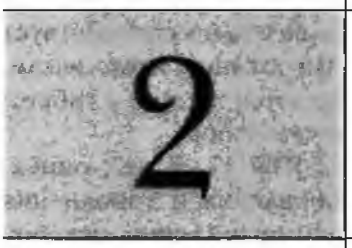
Еще один момент относительно эволюции Java: в 2006 году начался процесс открытия исходного кода Java. В наши дни доступны реализации JDK с открытым кодом. Открытие исходного кода дополнительно способствует динамической природе процесса разработки Java. В конечном итоге наследием инноваций Java является безопасность. Java остается живым и гибким языком, который привыкли ожидать в мире программирования.

Материал настоящей книги обновлен с учетом JDK 17. В ней описаны многочисленные функциональные средства, обновления и дополнения Java. Тем не менее, как подчеркивалось в предыдущем обсуждении, история программирования на Java характеризуется динамичными изменениями. Рекомендуется отслеживать новые возможности в каждом последующем выпуске Java. Проще говоря: эволюция Java продолжается!

Культура инноваций

Язык Java с самого начала находился в центре культуры инноваций. Его первый выпуск трансформировал подход к программированию для Интернета. Виртуальная машина Java (JVM) и байт-код поменяли представление о безопасности и переносимости. Переносимый код оживил веб-сеть. Процесс сообщества Java (Java Community Process — JCP) изменил способ внедрения новых идей в язык. Мир Java никогда не оставался на месте в течение долгого времени. JDK 17 — это последний выпуск в непрекращающейся, динамичной истории Java.

ГЛАВА



Краткий обзор языка Java

Как и в других языках программирования, элементы Java не существуют обособленно. Наоборот, они работают вместе, чтобы сформировать язык как единое целое. Однако такая взаимосвязанность может затруднить описание одного аспекта Java, не касаясь ряда остальных. Часто при обсуждении одной характеристики предполагается наличие знаний другой. По этой причине в главе представлен краткий обзор нескольких основных средств Java. Приведенный здесь материал послужит вам отправной точкой, которая позволит писать и понимать простые программы. Большинство обсуждаемых тем будут подробно рассматриваться в оставшихся главах части I.

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) заложено в саму основу Java. На самом деле все программы на Java являются в той или иной степени объектно-ориентированными. ООП настолько тесно связано с Java, что для написания даже самых простых программ на Java лучше изучить базовые принципы ООП. По этой причине глава начинается с обсуждения теоретических аспектов ООП.

Две парадигмы

Все компьютерные программы состоят из двух элементов: кода и данных. Кроме того, программа может быть концептуально организована вокруг своего кода или своих данных. Иными словами одни программы пишутся исходя из того, "что происходит", а другие — исходя из того, "что затронуто". Существуют две парадигмы, определяющие то, как строится программа. Первый способ называется *моделью, ориентированной на процессы.* Такой подход характеризует программу как последовательность линейных шагов (т.е. кода). Модель, ориентированную на процессы, можно рассматривать как *код, воздействующий на данные.* Процедурные языки, подобные C, успешно используют эту модель. Тем не менее, как упоминалось в главе 1, по мере роста размера и сложности программ при использовании этого подхода начинают возникать проблемы.

Для управления растущей сложностью был предложен второй подход, называемый *объектно-ориентированным программированием*. Объектно-ориентированное программирование позволяет организовать программу вокруг ее данных (т.е. объектов) и набора четко определенных интерфейсов к таким данным. Объектно-ориентированную программу можно охарактеризовать как *данные, управляющие доступом к коду*. Как будет показано, переключая управляющую сущность на данные, вы можете получить несколько организационных преимуществ.

Абстракция

Важнейшим элементом ООП является *абстракция*. Человеку свойственно справляться со сложностью через абстракцию. Например, люди не представляют себе автомобиль как набор из десятков тысяч отдельных деталей. Они думают о нем, как о четко определенном объекте со своим уникальным поведением. Такая абстракция позволяет людям доехать на автомобиле до продуктового магазина, не перегружаясь сложностью индивидуальных деталей. Они могут игнорировать подробности работы двигателя, коробки передач и тормозной системы. Взамен они могут свободно использовать объект как единое целое.

Эффективный способ управления абстракцией предусматривает применение иерархических классификаций. Они позволяют разбивать семантику сложных систем на более управляемые части. Снаружи автомобиль представляет собой единый объект. Но погрузившись внутрь, вы увидите, что автомобиль состоит из нескольких подсистем: рулевого управления, тормозов, аудиосистемы, ремней безопасности, обогрева, навигатора и т.д. Каждая подсистема в свою очередь содержит более специализированные узлы. Скажем, в состав аудиосистемы может входить радиоприемник, проигрыватель компакт-дисков и/или проигрыватель MP3. Суть в том, что вы управляете сложностью автомобиля (или любой другой сложной системы) за счет использования иерархических абстракций.

Иерархические абстракции сложных систем также можно применять к компьютерным программам. Данные из традиционной, ориентированной на процессы, программы могут быть трансформированы путем абстракции в составляющие ее объекты. Последовательность шагов процесса может стать совокупностью сообщений, передаваемых между этими объектами. Таким образом, каждый объект описывает свое уникальное поведение. Вы можете воспринимать такие объекты как конкретные сущности, которые реагируют на сообщения, указывающие им о необходимости *делать что-то*. В этом и заключается суть ООП.

Объектно-ориентированные концепции образуют основу Java точно так же, как они формируют базис человеческого понимания. Важно понимать, каким образом такие концепции воплощаются в программах. Вы увидите, что ООП является мощной и естественной парадигмой для создания программ, которые способны пережить неизбежные изменения, сопровождающие жизненный

цикл любого крупного программного проекта, в том числе осмысление, рост и устаревание. Например, когда у вас есть четко определенные объекты и чистые, надежные интерфейсы к этим объектам, вы можете без опасений элегантно выводить из эксплуатации или заменять части старой системы.

Три принципа ООП

Все языки ООП предоставляют механизмы, которые помогают реализовать объектно-ориентированную модель. Речь идет об инкапсуляции, наследовании и полиморфизме. Давайте взглянем на упомянутые концепции.

Инкапсуляция

Инкапсуляция представляет собой механизм, который связывает вместе код и обрабатываемые им данные, а также защищает их от внешнего вмешательства и неправильного использования. Инкапсуляцию можно считать защитной оболочкой, которая предотвращает произвольный доступ к коду и данным из другого кода, определенного вне оболочки. Доступ к коду и данным, находящимся внутри оболочки, строго контролируется через четко определенный интерфейс. Чтобы провести аналогию с реальным миром, рассмотрим автоматическую коробку передач автомобиля. Она инкапсулирует массу информации о вашем двигателе, такую как величина ускорения, наклон поверхности, по которой двигается автомобиль, и положение рычага переключения передач. Вы, как пользователь, располагаете только одним способом влияния на эту сложную инкапсуляцию: перемещение рычага переключения передач. Вы не можете воздействовать на коробку передач, скажем, с помощью сигнала поворота или дворников. Таким образом, рычаг переключения передач является четко определенным (и действительно уникальным) интерфейсом к коробке передач. Вдобавок то, что происходит внутри коробки передач, никак не влияет на объекты за ее пределами. Например, переключение передачи не включает фары! Поскольку автоматическая коробка передач инкапсулирована, десятки производителей автомобилей могут реализовать ее так, как им заблагорассудится. Однако с точки зрения водителя все они работают одинаково. Ту же самую идею можно применить и к программированию. Сила инкапсулированного кода в том, что каждый знает, как получить к нему доступ, и потому может использовать его независимо от деталей реализации и без каких-либо опасений столкнуться с неожиданными побочными эффектами.

Основой инкапсуляции в Java является класс. Хотя понятие класса подробно рассматривается позже в книге, полезно кратко обсудить его прямо сейчас. *Класс* определяет структуру и поведение (данные и код), которые будут общими для набора объектов. Каждый объект заданного класса содержит структуру и поведение, определенные классом, как если бы он был "отлит" в форме класса. По этой причине объекты иногда называют *экземплярами класса*. Таким образом, класс представляет собой логическую конструкцию, а объект имеет физическую реальность.

При создании класса вы указываете код и данные, составляющие класс. В совокупности такие элементы называются *членами* класса. В частности, данные, определенные классом, называют *переменными-членами* или *переменными экземпляра*. Код, работающий с этими данными, называют *методами-членами* или просто *методами*. (На тот случай, если вы знакомы с языком C или C++: то, что программист на Java называет *методом*, программист на C/C++ называет *функцией*.) В правильно написанных программах на Java методы определяют способ использования переменных-членов, т.е. поведение и интерфейс класса определяются методами, которые работают с данными его экземпляра.

Поскольку целью класса является инкапсуляция сложности, существуют механизмы для сокрытия сложности реализации внутри класса. Методы или переменные в классе могут быть помечены как закрытые или открытые. *Открытый* интерфейс класса представляет все, что должны или могут знать внешние пользователи класса. Доступ к *закрытым* методам и данным возможен только из кода, который является членом класса. Следовательно, любой другой код, не являющийся членом класса, не сможет получить доступ к закрытому методу или переменной. Так как доступ к закрытым членам класса другие части вашей программы могут получить только через открытые методы класса, вы можете гарантировать, что не будет совершено никаких неподходящих действий. Разумеется, это означает, что открытый интерфейс должен быть внимательно спроектирован, чтобы не раскрывать слишком много деталей внутренней работы класса (рис. 2.1).

Рис. 2.1. Инкапсуляция: открытые методы могут использовать для защиты закрытых данных

Наследование

Наследование представляет собой процесс, посредством которого один объект приобретает свойства другого объекта. Оно важно, т.к. поддерживает концепцию иерархической классификации. Ранее уже упоминалось, что большинство знаний стало доступным за счет иерархической (т.е. нисходящей) классификации. Например, золотистый ретривер является частью класса "собаки", который в свою очередь относится к классу "млекопитающие", входящему в состав более крупного класса "животные". В отсутствие иерархий каждый объект должен был бы явно определять все свои характеристики. Тем не менее, используя наследование, объекту нужно определить только те качества, которые делают его уникальным внутри своего класса. Он может наследовать общие атрибуты от своего родителя. Таким образом, именно механизм наследования позволяет одному объекту быть специфическим экземпляром более общего случая. Давайте подробнее рассмотрим сам процесс.

Большинство людей естественным образом воспринимают мир как состоящий из иерархически связанных друг с другом объектов, таких как животные, млекопитающие и собаки. При желании описать животных абстрактно вы бы сказали, что у них есть некоторые характерные признаки вроде размера, умственных способностей и типа костной системы. Животным также присущи определенные поведенческие аспекты: они едят, дышат и спят. Такое описание характерных признаков и поведения является определением класса для животных.

Если бы требовалось описать более конкретный класс животных, скажем, млекопитающих, то они имели бы более конкретные характерные признаки вроде типа зубов и молочных желез. Такое определение известно как подкласс животных, а животные являются *суперклассом* млекопитающих.

Поскольку млекопитающие — просто более точно определенные животные, они *наследуют* все характерные признаки животных. Находящийся глубоко в *иерархии классов* подкласс наследует все характерные признаки от каждого из своих предков (рис. 2.2).

Наследование также взаимодействует с инкапсуляцией. Если заданный класс инкапсулирует некоторые характерные признаки, тогда любой подкласс будет иметь те же самые признаки *плюс* любые, которые он добавляет как часть своей специализации (рис. 2.3). Это ключевая концепция, обеспечивающая возрастание сложности объектно-ориентированных программ линейно, а не геометрически. Новый подкласс наследует все характерные признаки всех своих предков. У него нет непредсказуемых взаимодействий с большей частью остального кода в системе.

Полиморфизм

Полиморфизм (от греческого "много форм") представляет собой средство, которое позволяет использовать один интерфейс для общего класса действий. Конкретное действие определяется природой ситуации. Возьмем в качестве примера стек (т.е. список, работающий по принципу "последним при-

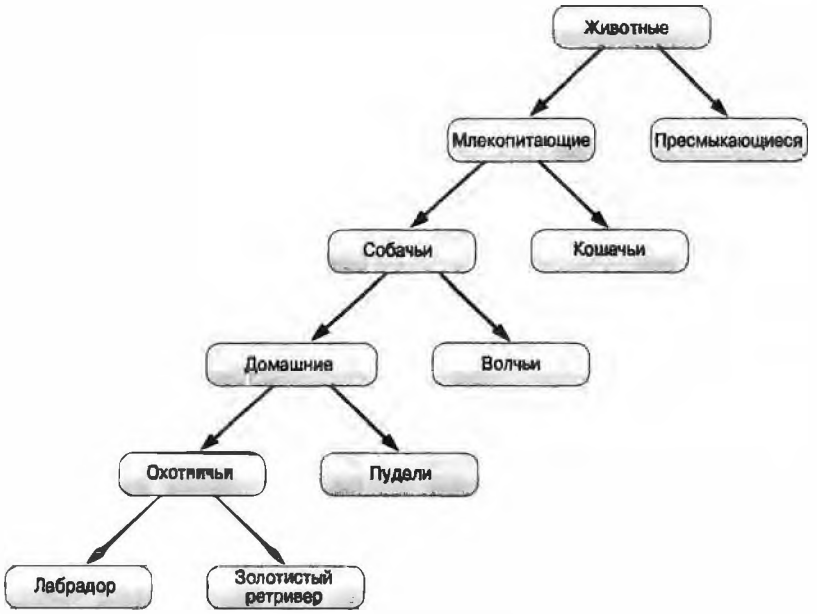


Рис. 2.2. Иерархия классов животных

шел — первым обслужен"). У вас может быть программа, требующая стеки трех типов. Один стек используется для целых значений, другой — для значений с плавающей точкой и третий — для символов. Каждый стек реализуется по тому же самому алгоритму, даже если хранящиеся данные различаются. В языке, не являющемся объектно-ориентированным, вам придется создать три разных набора стековых процедур с отличающимися именами. Но благодаря полиморфизму в Java вы можете указать общий набор стековых процедур с одинаковыми именами.

Как правило, концепция полиморфизма часто выражается фразой "один интерфейс, несколько методов". Это означает возможность разработки общего интерфейса для группы связанных действий, что поможет уменьшить сложность, позволив использовать один и тот же интерфейс для указания *общего класса действий*. Задачей компилятора будет выбор конкретного действия (т.е. метода) применительно к каждой ситуации. Вам, как программисту, не придется делать такой выбор вручную. Вам понадобится только запомнить и задействовать общий интерфейс.

Продолжая аналогию с собаками, можно отметить, что обоняние собаки полиморфно. Если собака почует кошку, то она залает и побежит за ней. Если собака почувствует запах еды, тогда у нее начнется слюноотделение, и она побежит к миске. В обеих ситуациях работает одно и то же обоняние. Разница в том, *что* именно издает запах, т.е. тип данных, с которыми имеет дело собачий нос! Та же общая концепция может быть реализована в Java, поскольку она применяется к методам внутри программы Java.

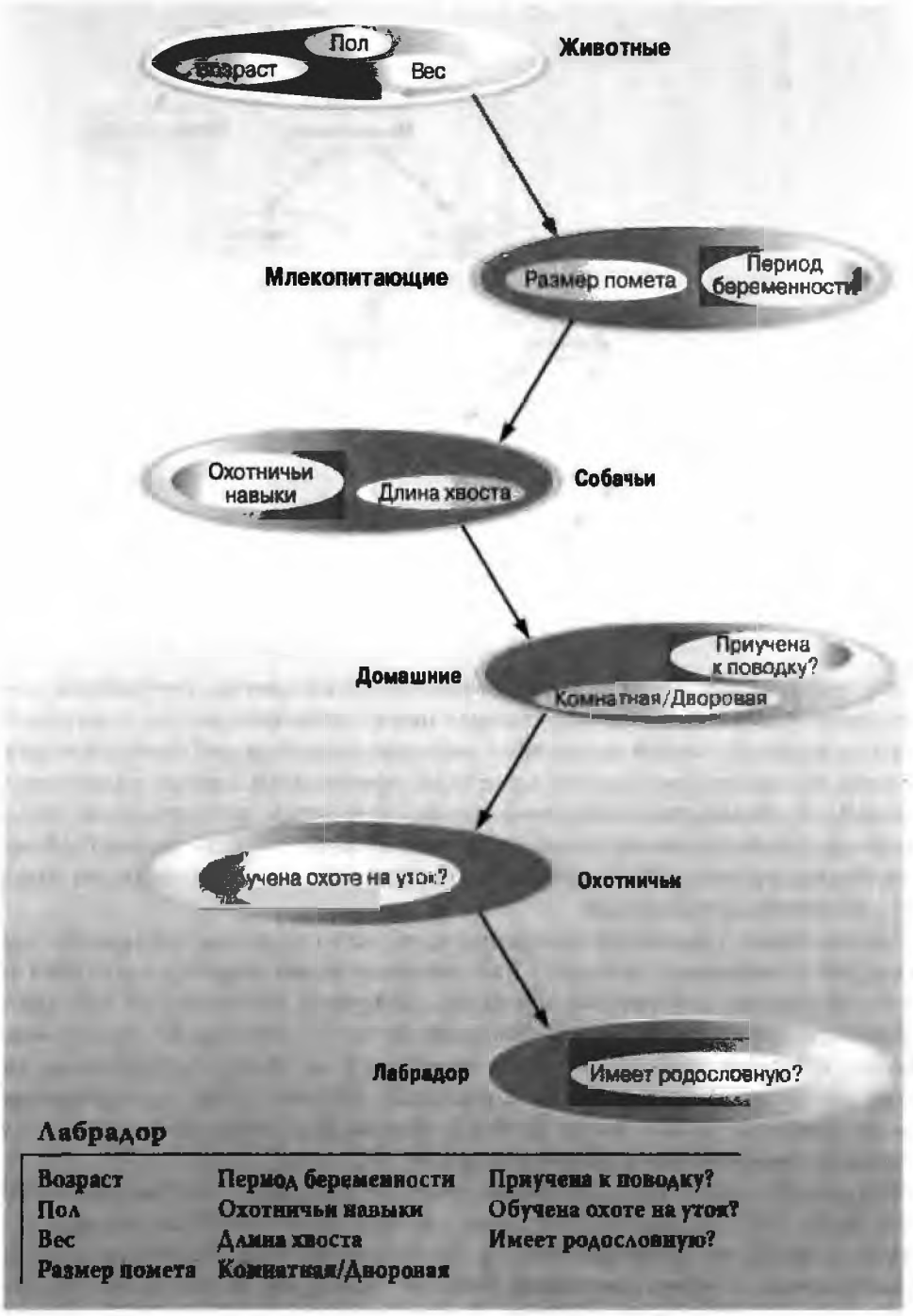


Рис. 2.3. Лабрадор наследует характерные признаки всех своих суперклассов

Совместная работа полиморфизма, инкапсуляции и наследования

При правильном применении полиморфизм, инкапсуляция и наследование объединяются для создания программной среды, которая поддерживает разработку гораздо более надежных и масштабируемых программ, чем в случае использования модели, ориентированной на процессы. Хорошо спроектированная иерархия классов является основой для многократного использования кода, в разработку и тестирование которого вы вложили время и усилия. Инкапсуляция позволяет вам со временем переносить свои реализации, не нарушая код, который зависит от открытого интерфейса ваших классов. Полиморфизм дает возможность создавать чистый, понятный, читабельный и устойчивый код.

Из двух реальных примеров автомобиль более полно иллюстрирует мощь ООП. Если о собаках интересно думать с точки зрения наследования, то автомобили больше похожи на программы. Все водители полагаются на наследование для управления различными типами (подклассами) транспортных средств. Независимо от того, является транспортное средство школьным автобусом, седаном "Мерседес", "Порше" или семейным минивэном, все водители могут более или менее находить и управлять рулем, педалью тормоза и педалью газа. Немного повозившись с рычагом переключения передач, большинство людей смогут даже отличить ручную коробку передач от автоматической, т.к. они в основном понимают их общий суперкласс — коробку передач.

Пользуясь автомобилями, люди постоянно взаимодействуют с их инкапсулированными характеристиками. Педали тормоза и газа скрывают невероятную сложность, а интерфейс настолько прост, что ими можно управлять с помощью ног! Реализация двигателя, тип тормозов и размер шин не влияют на то, каким образом вы взаимодействуете с определением класса педалей.

Последний принцип, полиморфизм, четко отражается в способности производителей автомобилей предлагать широкий спектр вариантов для одного и того же транспортного средства. Например, вы можете получить антиблокировочную или традиционную тормозную систему, рулевое управление с гидроусилителем или реечной передачей, а также 4-, 6- или 8-цилиндровый двигатель либо электромобиль. В любом случае вы по-прежнему будете нажимать на педаль тормоза для остановки, поворачивать руль для смены направления и нажимать на педаль газа, когда хотите двигаться. Один и тот же интерфейс может применяться для управления несколькими отличающимися реализациями.

Как видите, благодаря применению инкапсуляции, наследования и полиморфизма отдельные части трансформируются в объект, известный как автомобиль. То же самое относится и к компьютерным программам. За счет применения принципов ООП различные части сложной программы могут быть объединены в единое, надежное, сопровождаемое целое.

Как упоминалось в начале раздела, каждая программа на Java является объектно-ориентированной. Или, говоря точнее, каждая программа на Java включает в себя инкапсуляцию, наследование и полиморфизм. Хотя короткие примеры программ, приведенные в оставшейся части текущей главы и в ряде последующих глав, могут не выглядеть как обладающие всеми указанными характеристиками, тем не менее, эти характеристики присутствуют. Вы увидите, что многие возможности, предоставляемые языком Java, являются частью его встроенных библиотек классов, в которых широко используются инкапсуляция, наследование и полиморфизм.

Первая простая программа

После обсуждения объектно-ориентированного фундамента Java имеет смысл рассмотреть несколько фактических программ на Java. Давайте начнем с компиляции и запуска короткого примера программы, код которой показан ниже. Вы увидите, что это требует чуть большего объема работы, чем может показаться.

```
/*
  Простая программа на Java.
  Назовите этот файл Example.java.
*/
class Example {
  // Программа начинается с вызова main().
  public static void main(String[] args) {
    System.out.println("Простая программа на языке Java.");
  }
}
```

На заметку! В приведенных далее описаниях предполагается, что вы используете стандартный комплект разработчика Java SE Development Kit (JDK), предлагаемый компанией Oracle. (Доступны также версии с открытым кодом.) В случае применения интегрированной среды разработки (integrated development environment — IDE) вам придется следовать другой процедуре компиляции и запуска программ на Java. За подробными сведениями обращайтесь в документацию по IDE-среде.

Ввод кода программы

Для некоторых компьютерных языков имя файла, содержащего исходный код программы, не имеет значения. Однако с Java дело обстоит иначе. Первое, что вы должны узнать о Java — имя, которое вы назначаете файлу с исходным кодом, очень важно. В рассматриваемом примере именем файла с исходным кодом должно быть `Example.java` и вот причина.

Файл с исходным кодом в Java официально называется *единицей компиляции*. Он представляет собой текстовый файл, который содержит (помимо прочего) одно или большее число определений классов. (Пока будем использовать файлы с исходным кодом, содержащие только один класс.) Компилятор Java требует, чтобы в имени файла с исходным кодом применялось расширение `.java`.

Взглянув на программу, вы увидите, что именем определенного в ней класса является Example. Это не совпадение. В Java весь код должен находиться внутри класса. По соглашению имя главного класса должно совпадать с именем файла, содержащего программу. Вы также должны удостовериться в том, что прописные буквы в имени файла и в имени класса соответствуют друг другу. Причина связана с чувствительностью к регистру языка Java. В данный момент соглашение о том, что имена файлов соответствуют именам классов, может показаться деспотическим. Тем не менее, такое соглашение упрощает поддержку и организацию ваших программ. Более того, как вы увидите далее в книге, в некоторых случаях оно будет обязательным.

Компиляция программы

Чтобы скомпилировать программу Example, запустите компилятор javac, указав в командной строке имя файла с исходным кодом:

```
C:\>javac Example.java
```

Компилятор javac создает файл по имени Example.class с байт-кодом программы. Как обсуждалось ранее, байт-код Java является промежуточным представлением программы, содержащим инструкции, которые будет выполнять виртуальная машина Java (JVM). Таким образом, результатом javac не будет код, который можно запускать напрямую.

Чтобы действительно запустить программу, вам придется воспользоваться загрузчиком приложений Java под названием java. Для этого передайте ему в качестве аргумента командной строки имя Example:

```
C:\>java Example
```

Выполнение программы приводит к отображению следующего вывода:

```
Простая программа на языке Java.
```

Когда исходный код Java компилируется, каждый отдельный класс помещается в собственный выходной файл, имеющий имя класса и расширение .class. Вот почему рекомендуется назначать файлам с исходным кодом Java имя, совпадающее с именем класса, который они содержат — имя файла с исходным кодом будет совпадать с именем файла .class. При запуске java, как только что было показано, фактически указывается имя класса, который необходимо выполнить. Загрузчик приложений будет автоматически искать файл с таким именем и расширением .class. В случае нахождения файла он выполнит код, содержащийся в указанном классе.

На заметку! Начиная с JDK 11, в Java есть возможность запуска некоторых типов простых программ прямо из файла с исходным кодом без явного вызова javac. Такая методика может оказаться полезной в ряде ситуаций; она описана в приложении В. В настоящей книге предполагается, что вы применяете описанный выше нормальный процесс компиляции.

Подробный анализ первого примера программы

Несмотря на довольно небольшой размер программы `Example.java`, она содержит несколько основных характеристик, присущих всем программам на Java. Давайте займемся исследованием каждой части программы.

Программа начинается со следующих строк:

```
/*
  Простая программа на Java.
  Назовите этот файл Example.java.
*/
```

Строки представляют собой *комментарий*. Как и большинство других языков программирования, Java позволяет вводить примечания в файл с исходным кодом программы. Компилятор игнорирует содержимое комментариев. На самом деле комментарий описывает или объясняет работу программы любому, кто читает ее исходный код. В данном случае комментарий описывает программу и напоминает, что исходный файл должен называться `Example.java`. Конечно, в реальных приложениях комментарии обычно объясняют, как работает какая-то часть программы или что делает конкретная функция.

В Java поддерживаются три стиля комментариев. Комментарий в начале программы называется *многострочным*. Комментарии такого типа должны начинаться с символов `/*` и заканчиваться символами `*/`. Все, что находится между этими двумя парами символов, компилятор игнорирует. Как следует из названия, многострочный комментарий может занимать несколько строк.

Ниже показана следующая строка кода в программе:

```
class Example {
```

В строке с помощью ключевого слова `class` определяется новый класс. *Идентификатор* `Example` является именем класса. Все определение класса, включая всех его членов, будет находиться между открывающей фигурной скобкой ({) и закрывающей фигурной скобкой (}). В данный момент не слишком беспокойтесь о деталях класса помимо того, что в Java вся активность программы происходит внутри класса. Это одна из причин, по которой все программы на Java (по крайней мере, слегка) объектно-ориентированы.

Следующая строка в программе содержит *однострочный комментарий*:

```
// Программа начинается с вызова main().
```

Вы видите второй тип комментариев, поддерживаемых в Java. Однострочный комментарий начинается с символов `//` и простирается до конца строки. Как правило, программисты применяют многострочные комментарии для длинных примечаний, а однострочные — для кратких построчных описаний. Комментарии третьего типа, которые называются *документирующими*, обсуждается в разделе "Комментарии" далее в главе.

Вот следующая строка кода:

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

Данная строка начинает метод `main()`. Как объяснялось в предыдущем комментарии, с этой строки программа начнет выполняться. Обычно программа на Java начинает выполнение с вызова `main()`. Полностью осознать смысл каждой части строки пока невозможно, т.к. для этого нужно хорошо понимать подход Java к инкапсуляции. Однако поскольку такая строка кода присутствует в большинстве примеров в первой части книги, давайте кратко рассмотрим каждую часть.

Ключевое слово `public` представляет собой модификатор доступа, который позволяет программисту управлять видимостью членов класса. Когда член класса предварен ключевым словом `public`, доступ к нему может быть получен из кода за пределами класса, где он объявлен. (Противоположностью `public` является ключевое слово `private`, которое предотвращает использование члена кодом, определенным вне класса.) В данном случае метод `main()` должен быть объявлен как `public`, потому что при запуске программы его потребуется вызывать в коде за пределами класса. Ключевое слово `static` позволяет вызывать `main()` без создания конкретного экземпляра класса. Причина в том, что `main()` вызывается машиной JVM до создания каких-либо объектов. Ключевое слово `void` просто сообщает компилятору, что `main()` не возвращает значение. Как вы увидите, методы также могут возвращать значения. Если все это кажется немного запутанным, не переживайте — все концепции будут подробно рассмотрены в последующих главах.

Как уже упоминалось, метод `main()` вызывается при запуске приложения Java. Имейте в виду, что язык Java чувствителен к регистру, а потому `Main` отличается от `main`. Важно понимать, что компилятор Java будет компилировать классы, не содержащие метода `main()`. Но у `java` не будет возможности запускать такие классы. Таким образом, если вы наберете `Main` вместо `main`, то компилятор все равно скомпилирует вашу программу, но `java` сообщит об ошибке, поскольку не сможет найти метод `main()`.

Любая информация, которую вам нужно передать методу, получается переменными, указанными в наборе круглых скобок после имени метода. Такие переменные называются *параметрами*. Даже когда для метода не требуются параметры, вам все равно понадобится указать пустые круглые скобки. В `main()` всего один параметр, хоть и сложный. Конструкция `String[] args` объявляет параметр по имени `args`, который представляет собой массив экземпляров класса `String`. (Массивы — это совокупности похожих объектов.) Объекты типа `String` хранят строки символов. В данном случае `args` получает любые аргументы командной строки, присутствующие при выполнении программы. В рассматриваемой программе такая информация не используется, но другие программы, показанные далее в этой книге, будут ее потреблять.

Последним символом в строке является `{`, который сигнализирует о начале тела `main()`. Весь код, содержащийся в методе, будет находиться между открывающей и закрывающей фигурными скобками метода.

Еще один момент: `main()` — это просто стартовая точка для вашей программы. Сложная программа будет иметь десятки классов, только один из которых должен иметь метод `main()`, чтобы начать работу. Кроме того, для

некоторых типов программ метод `main()` вообще не нужен. Тем не менее, для большинства программ, приведенных в книге, метод `main()` обязателен.

Ниже показана следующая строка кода. Обратите внимание, что она расположена внутри `main()`.

```
System.out.println("Простая программа на языке Java.");
```

Здесь на экран выводится строка `"Простая программа на языке Java."` вместе с символом новой строки. Вывод в действительности осуществляется встроенным методом `println()`. В данном случае метод `println()` отображает переданную ему строку. В дальнейшем вы увидите, что `println()` можно применять и для отображения других типов информации. Строка начинается с `System.out`. Хотя подробно объяснить это сейчас слишком сложно, вкратце отметим, что `System` является предопределенным классом, обеспечивающим доступ к системе, а `out` — выходным потоком, подключенным к консоли.

Вероятно, вы уже догадались, что консольный вывод (и ввод) нечасто используется в большинстве реальных приложений Java. Поскольку большинство современных вычислительных сред являются графическими по своей природе, консольный ввод-вывод применяется в основном в простых утилитах, демонстрационных программах и серверном коде. Позже в книге вы узнаете о других способах генерирования вывода с использованием Java. Но пока мы продолжим применять методы консольного ввода-вывода.

Обратите внимание, что оператор `println()` завершается точкой с запятой. В Java точка с запятой присутствует в конце многих операторов. Как вы увидите, точка с запятой — важная часть синтаксиса Java.

Первый символ `}` в программе заканчивает метод `main()`, а последний символ `}` завершает определение класса `Example`.

Вторая простая программа

Вероятно, никакая другая концепция не является более фундаментальной для языка программирования, нежели переменная. Возможно, вам уже известно, что переменная представляет собой именованную ячейку памяти, которой ваша программа может присвоить значение. Значение переменной можно изменять во время выполнения программы. В следующей программе показано, как объявлять и присваивать значение переменной. В программе также иллюстрируется ряд новых аспектов консольного вывода. Как следует из комментариев в начале программы, файл потребуется назвать `Example2.java`.

```
/*
  Еще один короткий пример.
  Назовите этот файл Example2.java.
*/
class Example2 {
  public static void main(String[] args) {
    int num;    // объявление переменной по имени num

    num = 100;  // присваивание переменной num значения 100

    System.out.println("Значение num: " + num);
```

```
    num = num * 2;
    System.out.print("Значение num * 2: ");
    System.out.println(num);
  }
}
```

Запустив программу, вы получите следующий вывод:

```
Значение num: 100
Значение num * 2: 200
```

Давайте выясним, почему генерируется такой вывод. Ниже приведена новая строка в программе:

```
int num;    // объявление переменной по имени num
```

В строке объявляется целочисленная переменная по имени num. Подобно многим другим языкам переменные в Java до своего использования должны быть объявлены.

Вот как выглядит общая форма объявления переменной:

```
тип имя-переменной;
```

Здесь тип указывает тип объявляемой переменной, а имя-переменной — имя переменной. При желании объявить более одной переменной заданного типа можете применять список имен переменных, отделенных друг от друга запятыми. В Java определено несколько типов данных, включая целочисленный, символьный и числовой с плавающей точкой. Целочисленный тип указывается с помощью ключевого слова int. Следующая строка в программе:

```
num = 100;  // присваивание переменной num значения 100
```

обеспечивает присваивание переменной num значения 100. Операция присваивания в Java обозначается одиночным знаком равенства.

В показанной далее строке кода выводится значение num, предваренное строкой "Значение num:".

```
System.out.println("Значение num: " + num);
```

Знак + в операторе приводит к тому, что значение num добавляется к строке, которая ему предшествует, после чего результирующая строка выводится. (На самом деле значение num сначала преобразуется из целочисленного в эквивалентное строковое и затем объединяется с предшествующей ему строкой. Этот процесс подробно описан далее в книге.) Такой подход можно обобщить. Используя операцию +, вы можете объединять в одном операторе println() столько элементов, сколько хотите.

В следующей строке кода переменной num присваивается значение num, умноженное на 2. Как и в большинстве других языков, операция умножения в Java обозначается символом *. После выполнения строки кода переменная num будет содержать значение 200.

Ниже показаны очередные две строки программы:

```
System.out.print ("Значение num * 2: ");
System.out.println (num);
```

Здесь происходит несколько новых вещей. Первым делом с применением встроенного метода `print()` отображается строка `"Значение num * 2: "`. Она не заканчивается символом новой строки, т.е. вывод, генерируемый следующим, будет начинаться в той же самой строке. Метод `print()` аналогичен методу `println()`, но он не выводит символ новой строки после каждого вызова. Взгляните теперь на вызов `println()`. Обратите внимание, что переменная `num` используется сама по себе. Методы `print()` и `println()` можно применять для вывода значений любых встроенных типов Java.

Два управляющих оператора

Хотя управляющие операторы будут подробно рассматриваться в главе 5, здесь кратко представлены два из них, чтобы их можно было использовать в примерах программ, приведенных в главах 3 и 4. Они также помогут проиллюстрировать важный аспект Java: блоки кода.

Оператор `if`

Оператор `if` в Java работает во многом аналогично условному оператору в любом другом языке. Он определяет поток выполнения на основе того, является некоторое условие истинным или ложным. Ниже показана его простейшая форма:

```
if(условие) оператор;
```

Здесь условие — это булевское выражение. (Булевским является такое выражение, результатом вычисления которого будет либо `true` (истина), либо `false` (ложь).) Если условие истинно, тогда оператор выполняется. Если условие ложно, то оператор пропускается. Вот пример:

```
if(num < 100) System.out.println("Значение num меньше 100");
```

В данном случае, если переменная `num` содержит значение, которое меньше 100, тогда условное выражение дает `true` и вызов метода `println()` выполняется. Если же `num` содержит значение, которое больше или равно 100, то вызов метода `println()` пропускается.

Как будет показано в главе 4, в Java определен полный набор операций отношения, которые можно применять в условном выражении. В табл. 2.1 описано несколько из них.

Таблица 2.1. Часто используемые операции отношения

Операция	Описание
<	Меньше
>	Больше
==	Равно

Обратите внимание, что проверка на равенство обозначается двумя знаками равенства. Ниже приведена программа, в которой иллюстрируется работа оператора `if`:

```
/*
  Демонстрация работы оператора if.
  Назовите этот файл IfSample.java.
*/
class IfSample {
  public static void main(String[] args) {
    int x, y;
    x = 10;
    y = 20;
    if(x < y) System.out.println("Значение x меньше y");

    x = x * 2;
    if(x == y) System.out.println("Теперь значение x равно y");

    x = x * 2;
    if(x > y) System.out.println("Теперь значение x больше y");

    // Здесь ничего не отобразится
    if(x == y) System.out.println("Этот вывод вы не увидите");
  }
}
```

В результате выполнения программы генерируется следующий вывод:

```
Значение x меньше y
Теперь значение x равно y
Теперь значение x больше y
```

С программой связан еще один момент. В строке

```
int x, y;
```

объявляются две переменные, x и y, посредством списка с разделителем-запятой.

Цикл for

Операторы циклов — важная часть почти любого языка программирования, поскольку они обеспечивают возможность многократного выполнения некоторой задачи. Как вы увидите в главе 5, язык Java предлагает мощный набор конструкций циклов. Возможно, наиболее универсальным является цикл `for`. Вот простейшая форма цикла `for`:

```
for(инициализация; условие; итерация) оператор;
```

В своей самой распространенной форме часть `инициализация` цикла устанавливает переменную управления циклом в начальное значение. Часть `условие` представляет собой булевское выражение, которое проверяет переменную управления циклом. Если результат проверки оказывается истинным, тогда `оператор` выполняется, а цикл `for` продолжает работу. При ложном результате проверки цикл завершается. Выражение `итерация` определяет, каким образом переменная управления циклом изменяется на каждой итерации цикла. Далее приведена короткая программа, иллюстрирующая работу цикла `for`:

```
/*
  Демонстрация работы цикла for.
  Назовите этот файл ForTest.java.
*/
```

```
class ForTest {
  public static void main(String[] args) {
    int x;

    for(x = 0; x<10; x = x+1)
      System.out.println("Значение x: " + x);
  }
}
```

Программа генерирует такой вывод:

```
Значение x: 0
Значение x: 1
Значение x: 2
Значение x: 3
Значение x: 4
Значение x: 5
Значение x: 6
Значение x: 7
Значение x: 8
Значение x: 9
```

В данном примере x — это переменная управления циклом. Она инициализируется нулем в части инициализация цикла `for`. В начале каждой итерации (включая первую) выполняется условная проверка `x<10`. Если результат проверки оказывается истинным, тогда выполняется оператор `println()`, после чего выполняется часть итерация цикла, которая увеличивает значение x на 1. Процесс продолжается до тех пор, пока условная проверка не станет ложной.

Интересно отметить, что в профессионально написанных программах на Java вы практически никогда не встретите часть итерация цикла, написанную так, как в предыдущей программе. То есть вы будете редко видеть операторы вроде следующего:

```
x = x + 1;
```

Причина в том, что в Java есть специальная операция инкремента, обладающая большей эффективностью, которая обозначается посредством ++ (т.е. два знака "плюс" подряд). Операция инкремента увеличивает свой операнд на единицу. С помощью операции инкремента предыдущее выражение можно записать в показанной ниже форме:

```
x++;
```

Таким образом, цикл `for` в предыдущей программе обычно будет записываться в следующем виде:

```
for(x = 0; x<10; x++)
```

Можете опробовать его. Вы заметите, что цикл выполняется в точности, как было ранее.

В Java также предлагается операция декремента, обозначаемая как --. Она уменьшает свой операнд на единицу.

Использование блоков кода

Язык Java позволяет группировать два или более операторов в *блоки кода*, также называемые *кодовыми блоками*. Для этого операторы помещаются между открывающей и закрывающей фигурными скобками. После того, как блок кода создан, он становится логической единицей, которую можно применять в любом месте, где разрешено использовать одиночный оператор. Скажем, блок может служить целью для операторов `if` и `for`. Возьмем следующий оператор `if`:

```
if(x < y) {     // начало блока
  x = y;
  y = 0;
}               // конец блока
```

Если `x` меньше `y`, тогда выполнятся оба оператора внутри блока. Таким образом, два оператора внутри блока образуют логическую единицу, где первый оператор не может быть выполнен без выполнения второго. Ключевой момент здесь в том, что всякий раз, когда нужно логически связать два или большее количество операторов, вы создаете блок.

Давайте рассмотрим еще один пример. В показанной далее программе блок кода применятся в качестве цели цикла `for`.

```
/*
  Демонстрация работы блока кода.
  Назовите этот файл BlockTest.java.
*/
class BlockTest {
  public static void main(String[] args) {
    int x, y;
    y = 20;
    // целью этого цикла является блок
    for(x = 0; x<10; x++) {
      System.out.println("Значение x: " + x);
      System.out.println("Значение y: " + y);
      y = y - 2;
    }
  }
}
```

Программа генерирует следующий вывод:

```
Значение x: 0
Значение y: 20
Значение x: 1
Значение y: 18
Значение x: 2
Значение y: 16
Значение x: 3
Значение y: 14
Значение x: 4
Значение y: 12
Значение x: 5
Значение y: 10
Значение x: 6
```

```
Значение y: 8
Значение x: 7
Значение y: 6
Значение x: 8
Значение y: 4
Значение x: 9
Значение y: 2
```

В этом случае целью цикла `for` является блок кода, а не одиночный оператор. Соответственно на каждой итерации цикла будут выполняться три оператора внутри блока, о чем, конечно же, свидетельствует вывод, генерируемый программой.

Позже в книге вы увидите, что блоки кода обладают дополнительными характеристиками и сценариями использования. Однако главная причина их существования — создание логически неразрывных единиц кода.

Лексические вопросы

После ознакомления с несколькими короткими программами на Java наступило время более формально описать атомарные элементы Java. Программы на Java представляют собой совокупность пробельных символов, идентификаторов, литералов, комментариев, операторов, разделителей и ключевых слов. Операторы обсуждаются в следующей главе, остальное же описано далее.

Пробельные символы

Java — язык свободной формы, т.е. вы не обязаны следовать каким-то особым правилам в отношении отступов. Например, программу `Example` можно было бы записать целиком в одной строке или любым другим странным способом при условии наличия хотя бы одного пробельного символа между каждой парой лексем, которые еще не были разграничены операцией или разделителем. В языке Java к пробельным символам относятся пробел, табуляция, новая строка или перевод страницы.

Идентификаторы

Идентификаторы применяются для именования таких вещей, как классы, переменные и методы. Идентификатором может быть любая описательная последовательность прописных и строчных букв, цифр или символов подчеркивания и знака доллара. (Знак доллара не предназначен для общего использования.) Они не должны начинаться с цифры, чтобы компилятор не путал их с числовым литералом. Как вы помните, язык Java чувствителен к регистру, поэтому `VALUE` и `Value` — разные идентификаторы. Вот несколько примеров допустимых идентификаторов:

```
AvgTemp     count       a4          $test       this_is_ok
```

Ниже представлены примеры недопустимых имен идентификаторов:

```
2count      high-temp   Not/ok
```

На заметку! Начиная с JDK 9, одиночный символ подчеркивания нельзя применять в качестве идентификатора.

Литералы

Константное значение в Java создается с использованием его *литерального* представления. Ниже приведены примеры литералов:

```
100        98.6        'X'          "Тестовая строка"
```

Начиная слева, первый литерал задает целочисленное значение, второй — значение с плавающей точкой, третий — символьную константу и четвертый — строковое значение. Литерал можно применять везде, где допускается значение его типа.

Комментарии

Ранее уже упоминалось, что в Java определены три типа комментариев. Два типа вы видели: однострочный и многострочный. Третий тип называется *документирующим комментарием* и используется для создания HTML-файла, который документирует вашу программу. Документирующий комментарий начинается с символов `/**` и заканчивается символами `*/`. Документирующие комментарии обсуждаются в приложении А.

Разделители

В Java есть несколько символов, которые применяются в качестве разделителей. Наиболее часто используемый разделитель в Java — точка с запятой. Как вы видели, он часто применяется для завершения операторов. Допустимые разделители описаны в табл. 2.2.

Ключевые слова Java

В настоящее время в языке Java определено 67 ключевых слов (табл. 2.3). В сочетании с синтаксисом операторов и разделителей они формируют основу языка Java. Как правило, ключевые слова нельзя применять в качестве идентификаторов, т.е. они не могут использоваться в качестве имен для переменных, классов или методов. Тем не менее, 16 ключевых слов являются контекстно-чувствительными, а это значит, что они будут служить ключевыми словами только в случае применения с функциональным средством, к которому относятся. Они поддерживают функциональные средства, появившиеся в Java за последние несколько лет. Десять ключевых слов относятся к модулям: `exports`, `module`, `open`, `opens`, `provides`, `requires`, `to`, `transitive`, `uses` и `with`. Записи объявляются с помощью `record`. Для запечатанных классов и интерфейсов используется `sealed`, `non-sealed` и `permits`; `yield` применяется с расширенным оператором `switch`; `var` поддерживает выведение типов локальных переменных. Поскольку они зависят от контекста, их добавление не повлияло на существующие программы.

Таблица 2.2. Символы, используемые в качестве разделителей

Символ	Название	Описание
`()`	Круглые скобки	Применяется для указания списков параметров в определениях и вызовах методов. Кроме того, используется для определения порядка выполнения операций в обычных выражениях, выражениях внутри управляющих операторов и при приведении типов
`{}`	Фигурные скобки	Применяется для указания значений автоматически инициализируемых массивов. Также используется для определения блоков кода, классов, методов и локальных областей действия
`[]`	Квадратные скобки	Применяется для объявления типов массивов. Кроме того, используется для разыменования значений массива
`;`	Точка с запятой	Завершает операторы
`,`	Запятая	Отделяет последовательно следующие друг за другом идентификаторы при объявлении переменных. Также применяется для объединения операторов внутри `for`
`.`	Точка	Используется для отделения имен пакетов от имен подпакетов и классов. Кроме того, применяется для отделения имени переменной или метода от имени ссылочной переменной
`::`	Двоеточия	Используется для создания ссылки на метод или конструктор
`...`	Троеточие	Указывает параметр с переменным количеством аргументов
`@`	Коммерческое "эт" (символ "а" с тонким спиральным штрихом)	Начинает аннотацию

Кроме того, начиная с JDK 9, подчеркивание само по себе считается ключевым словом, чтобы предотвратить его использование в качестве имени чего-либо в программе. Начиная с JDK 17, ключевое слово `strictfp` объявлено устаревшим, т.е. не имеет никакого эффекта.

Ключевые слова const и goto зарезервированы, но не применяются. На заре развития Java несколько других ключевых слов были зарезервированы для возможного использования в будущем. Однако в текущей спецификации Java определены только ключевые слова, перечисленные в табл. 2.3.

Таблица 2.3. Ключевые слова Java

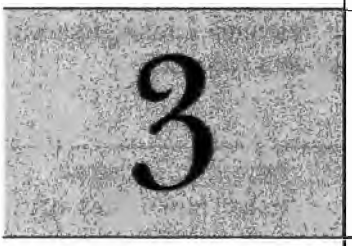
abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	exports	extends
final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long
module	native	new	non-sealed	open	opens
package	permits	private	protected	provides	public
record	requires	return	sealed	short	static
strictfp	super	switch	synchronized	this	throw
throws	to	transient	transitive	try	uses
var	void	volatile	while	with	yield
_					

Помимо ключевых слов в Java зарезервированы еще три имени, которые были частью Java с самого начала: true, false и null.Они представляют собой значения, определенные в Java, и не могут применяться в качестве имен переменных, классов и т.д.

Библиотеки классов Java

В примерах программ, приведенных в главе, использовались два встроенных метода Java: println() и print(). Как упоминалось ранее, указанные методы доступны через System.out. Здесь System — это предопределенный класс Java, который автоматически включается в ваши программы. В более широком плане среда Java опирается на несколько встроенных библиотек классов, которые содержат множество встроенных методов, обеспечивающих поддержку таких средств, как ввод-вывод, обработка строк, работа с сетью и графика. Стандартные классы также обеспечивают поддержку графического пользовательского интерфейса. Таким образом, Java как совокупность представляет собой сочетание самого языка Java с его стандартными классами. Позже вы увидите, что библиотеки классов предоставляют большую часть функциональности, связанной с Java. Действительно, частью становления программиста на Java является обучение использованию стандартных классов Java. В части I книги по мере необходимости описываются различные элементы классов и методов стандартной библиотеки. В части II подробно рассматриваются несколько библиотек классов.

ГЛАВА



Типы данных, переменные и массивы

В настоящей главе рассматриваются три наиболее фундаментальных элемента Java: типы данных, переменные и массивы. Как и все современные языки программирования, в Java поддерживается несколько типов данных. Вы можете использовать такие типы для объявления переменных и создания массивов. Далее вы увидите, что подход Java к этим элементам может отличается ясностью, эффективностью и связностью.

Java — строго типизированный язык

Первым делом важно отметить, что Java является строго типизированным языком. Действительно, отчасти безопасность и надежность Java проистекают как раз из данного факта. Давайте посмотрим, что это значит. Во-первых, у каждой переменной есть тип, у каждого выражения есть тип, и каждый тип строго определен. Во-вторых, все присваивания, как явные, так и через передачу параметров в вызовах методов, проверяются на совместимость типов. Автоматическое приведение или преобразование конфликтующих типов, как делается в некоторых языках, отсутствует. Компилятор Java проверяет все выражения и параметры для гарантирования совместимости типов. Любые несоответствия типов считаются ошибками, которые должны быть исправлены до того, как компилятор завершит компиляцию класса.

Примитивные типы

В Java определены восемь *примитивных* типов данных: `byte`, `short`, `int`, `long`, `char`, `float`, `double` и `boolean`. Примитивные типы также часто называют *простыми* типами, и в книге будут применяться оба термина. Их можно разделить на четыре группы.

- **Целые числа.** Эта группа включает типы `byte`, `short`, `int` и `long`, предназначенные для представления целых чисел со знаком.
- **Числа с плавающей точкой.** В эту группу входят типы `float` и `double`, которые представляют числа с точностью до определенного знака после десятичной точки.

- **Символы.** Эта группа включает тип `char`, предназначенный для представления символов из набора наподобие букв и цифр.
- **Булевские значения.** В эту группу входит тип `boolean`, который является специальным типом, представляющим истинные и ложные значения.

Вы можете использовать перечисленные выше типы в том виде, как есть, либо создавать массивы или собственные типы классов. Таким образом, они образуют основу для всех других типов данных, которые вы можете создать.

Примитивные типы представляют одиночные значения, а не сложные объекты. Хотя в остальном язык Java полностью объектно-ориентирован, примитивные типы — нет. Они аналогичны простым типам, которые встречаются в большинстве других не объектно-ориентированных языков. Причиной кроется в эффективности. Превращение примитивных типов в объекты слишком сильно снизило бы производительность.

Примитивные типы определены так, чтобы иметь явный диапазон и обладать математически строгим поведением. Языки вроде C и C++ позволяют изменять размер целого числа в зависимости от требований среды выполнения. Однако Java в этом отношении отличается. Из-за требования переносимости Java все типы данных имеют строго определенный диапазон. Например, `int` всегда занимает 32 бита независимо от конкретной платформы. В итоге появляется возможность написания программ, которые гарантированно будут работать *без переноса* на любую машинную архитектуру. Хотя строгое установление размера целого числа в некоторых средах может привести к небольшой потере производительности, оно необходимо для обеспечения переносимости.

Давайте взглянем на каждый тип по очереди.

Целые числа

В Java определены четыре целочисленных типа: `byte`, `short`, `int` и `long`. Все они представляют положительные и отрицательные значения. В Java не поддерживаются только положительные целые числа без знака. Во многих других языках программирования поддерживаются как целые числа со знаком, так и целые числа без знака, но разработчики Java решили, что целые числа без знака не нужны. В частности, они считали, что понятие *без знака* использовалось в основном для указания поведения *старшего бита*, который определяет *знак* целочисленного значения. В главе 4 вы увидите, что Java по-другому управляет смыслом старшего бита, добавляя специальную операцию "беззнакового сдвига вправо". Таким образом, необходимость в целочисленном типе без знака попросту отпала.

Ширина (или разрядность) целочисленного типа не должна трактоваться как объем потребляемой им памяти, а скорее как *поведение*, которое он определяет для переменных и выражений данного типа. Исполняющая среда Java может свободно использовать любой желаемый размер при условии, что типы ведут себя так, как вы их объявили. Ширина и диапазоны целочисленных типов сильно различаются, как показано в табл. 3.1.

Таблица 3.1. Разрядность и диапазоны целочисленных типов

Имя	Ширина в битах	Диапазон
long	64	От –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
int	32	От –2 147 483 648 до 2 147 483 647
short	16	От –32 768 до 32 767
byte	8	От –128 до 127

А теперь более подробно рассмотрим каждый целочисленный тип.

Тип byte

Наименьшим целочисленным типом является byte. Он является 8-битным типом со знаком, диапазон значений которого составляет от –128 до 127. Переменные типа byte особенно удобны при работе с потоком данных из сети или файла. Они также полезны, когда приходится иметь дело с низкоуровневыми двоичными данными, которые могут быть несовместимыми напрямую с другими встроенными типами Java.

Байтовые переменные объявляются с применением ключевого слова byte. Например, ниже объявлены две переменные типа byte с именами b и c:

```
byte b, c;
```

Тип short

Тип short представляет собой 16-битный тип с диапазоном значений от –32 768 до 32 767. Он является, вероятно, наименее часто используемым типом в Java. Вот несколько примеров объявлений переменных типа short:

```
short s;
short t;
```

Тип int

Самым распространенным целочисленным типом можно считать int — 32-битный тип, диапазон значений которого составляет от –2 147 483 648 до 2 147 483 647. Помимо других целей переменные типа int обычно применяются для управления циклами и для индексации массивов. Хотя может показаться, что использование типа byte или short будет эффективнее int в ситуациях, когда больший диапазон int не нужен, это не всегда так. Причина в том, что когда в выражении присутствуют значения byte и short, при вычислении выражения они повышаются до int. (Повышение типов обсуждается позже в главе.) Таким образом, int часто является лучшим выбором, когда требуется целое число.

Тип long

Тип long представляет собой 64-битный тип со знаком и полезен в случаях, когда тип int недостаточно велик для хранения желаемого значения. Диапазон значений long довольно широк, что делает его удобным типом для работы с крупными целыми числами. Скажем, вот программа, которая рассчитывает количество миль, проходимых светом за указанное количество дней:

```
// Рассчитать расстояние, проходимое светом,
// с применением переменных типа long.
class Light {
  public static void main(String[] args) {
    int lightspeed;
    long days;
    long seconds;
    long distance;

    // Приблизительная скорость света в милях за секунду.
    lightspeed = 186000;
    days = 1000;                        // указать количество дней
    seconds = days * 24 * 60 * 60;      // преобразовать в секунды
    distance = lightspeed * seconds;    // рассчитать расстояние

    // Вывести примерное расстояние в милях, проходимое светом
    // за указанное количество дней.
    System.out.print("За " + days);
    System.out.print(" дней свет пройдет около ");
    System.out.println(distance + " миль.");
  }
}
```

Программа сгенерирует следующий вывод:

```
За 1000 дней свет пройдет около 16070400000000 миль.
```

Совершенно очевидно, что результат не уместился бы в переменную типа int.

Типы с плавающей точкой

Числа с плавающей точкой, также известные как *вещественные* числа, используются при вычислении выражений, требующих точности до определенного знака после десятичной точки. Например, вычисления вроде квадратного корня или трансцендентных функций, подобных синусу и косинусу, дают в результате значение, точность которого требует типа с плавающей точкой. В Java реализован стандартный (IEEE-754) набор типов и операций с плавающей точкой. Существуют две разновидности типов с плавающей точкой, float и double, которые представляют числа с одинарной и двойной точностью соответственно. Их ширина и диапазоны приведены в табл. 3.2.

Таблица 3.2. Разрядность и диапазоны типов с плавающей точкой

Имя	Ширина в битах	Приблизительный диапазон
double	64	От 4,9e–324 до 1,8e+308
float	32	От 1,4e–045 до 3,4e+038

Ниже каждый тип рассматривается по отдельности.

Тип float

Тип float определяет значение *одинарной точности*, которое занимает 32 бита памяти. На некоторых процессорах значения одинарной точности обрабатываются быстрее и занимают вдвое меньше места, чем значения двойной точности, но вычисления становятся неточными при очень больших или очень маленьких значениях. Переменные типа float удобны, когда нужен дробный компонент, но не требуется высокая точность. Например, тип float может быть полезен при представлении сумм в долларах и центах.

Вот несколько примеров объявлений переменных типа float:

```
float hightemp, lowtemp;
```

Тип double

Значение *двойной точности*, обозначаемое ключевым словом double, занимает 64 бита. На некоторых современных процессорах, оптимизированных под высокоскоростные математические вычисления, обработка значений двойной точности в действительности выполняется быстрее, чем значений одинарной точности. Все трансцендентные математические функции, такие как sin(), cos() и sqrt(), возвращают двойные значения. Когда вам нужно поддерживать точность в течение многократно повторяющихся вычислений или манипулировать числами с крупными значениями, то тип double будет наилучшим вариантом.

Ниже показана короткая программа, в которой переменные типа double применяются для вычисления площади круга:

```
// Вычислить площадь круга.
class Area {
  public static void main(String[] args) {
    double pi, r, a;

    r = 10.8;             // радиус круга
    pi = 3.1416;          // приближенное значение pi
    a = pi * r * r;       // вычислить площадь

    System.out.println("Площадь круга равна " + a);
  }
}
```

Символы

Для хранения символов в Java используется тип данных char. Важно понять один ключевой момент: для представления символов в Java применяется Unicode. Кодировка Unicode определяет полностью международный набор символов, с помощью которого можно представить все символы, встречающиеся во всех естественных языках. Он объединяет десятки наборов символов, таких как романский, греческий, арабский, кириллический, иврит, катакана, хангул и многие другие. Во время создания Java для Unicode требовалось 16 бит. Таким образом, в Java тип char является 16-битным с диапазоном значений от 0 до 65 535. Отрицательных значений char не бывает. Стандартный набор символов, известный как ASCII, по-прежнему находится в диапазоне от 0 до 127, а расширенный 8-битный набор символов, ISO-Latin-1 — в диапазоне от 0 до 255. Поскольку язык Java предназначен для написания программ, используемых по всему миру, при представлении символов имеет смысл применять Unicode. Конечно, использование Unicode несколько неэффективно для таких языков, как английский, немецкий, испанский или французский, символы которых могут легко умещаться в пределах 8 бит. Но такова цена, которую приходится платить за глобальную переносимость.

На заметку! Дополнительную информацию о Unicode можно найти на веб-сайте `http://www.unicode.org`.

В следующей программе демонстрируется применение переменных char:

```
// Демонстрация использования типа данных char.
class CharDemo {
  public static void main(String[] args) {
    char ch1, ch2;
    ch1 = 88;  // код для X
    ch2 = 'Y';
    System.out.print("ch1 и ch2: ");
    System.out.println(ch1 + " " + ch2);
  }
}
```

Вот вывод, отображаемый в результате выполнения программы:

```
ch1 и ch2: X Y
```

Обратите внимание, что переменной ch1 присвоено значение 88, которое является кодом ASCII (и Unicode), соответствующим букве X. Как уже упоминалось, набор символов ASCII занимает первые 127 значений в наборе символов Unicode. По этой причине все "старые уловки", которые вы могли использовать с символами в других языках, будут работать и в Java.

Хотя тип char предназначен для хранения символов Unicode, его также можно применять как целочисленный тип, с которым допускается выполнять арифметические операции. Скажем, вы можете сложить два символа вместе либо инкрементировать значение символьной переменной. Взгляните на приведенную далее программу:

```
// Переменные char ведут себя подобно целым числам.
class CharDemo2 {
  public static void main(String[] args) {
    char ch1;

    ch1 = 'X';
    System.out.println("ch1 содержит " + ch1);

    ch1++;  // инкрементировать ch1
    System.out.println("ch1 теперь содержит " + ch1);
  }
}
```

Программа генерирует показанный ниже вывод:

```
ch1 содержит X
ch1 теперь содержит Y
```

В программе переменной `ch1` сначала присваивается значение `'X'`. Затем выполняется инкрементирование `ch1`. В результате `ch1` содержит значение `'Y'`, т.е. следующий символ в последовательности ASCII (и Unicode).

На заметку! В формальной спецификации Java тип `char` упоминается как *целочисленный тип*, а это значит, что он относится к той же общей категории, куда входят `int`, `short`, `long` и `byte`. Но поскольку тип `char` используется в основном для представления символов Unicode, обычно он считается отдельной категорией.

Булевские значения

В Java имеется примитивный тип под названием `boolean`, предназначенный для хранения булевских значений. Возможных значений только два: `true` и `false`. Значения такого типа возвращаются всеми операциями отношения, например, `a<b`. Использование типа `boolean` также обязательно в условных выражениях в операторах управления, таких как `if` и `for`. Ниже показана программа, в которой демонстрируется применение типа `boolean`:

```
// Демонстрация использования значений boolean.
class BoolTest {
  public static void main(String[] args) {
    boolean b;

    b = false;
    System.out.println("b равно " + b);
    b = true;
    System.out.println("b равно " + b);

    // Значение boolean может управлять оператором if.
    if(b) System.out.println("Это выполняется.");

    b = false;
    if(b) System.out.println("Это не выполняется.");

    // Результатом операции отношения является значение boolean.
    System.out.println("10 > 9 равно " + (10 > 9));
  }
}
```

Программа генерирует такой вывод:

```
b равно false
b равно true
Это выполняется.
10 > 9 равно true
```

В этой программе следует отметить три интересных момента. Во-первых, как видите, когда `println()` выводит булевское значение, отображается `true` или `false`. Во-вторых, самого по себе значения булевской переменной достаточно для управления оператором `if`, т.е. нет необходимости записывать оператор `if` в виде:

```
if(b == true) ...
```

В-третьих, результатом операции отношения, подобной <, является булевское значение. Именно потому выражение `10>9` отображает значение `true`. Кроме того, дополнительный набор скобок вокруг `10>9` нужен из-за того, что приоритет операции + выше приоритета операции >.

Подробный анализ литералов

Литералы кратко упоминались в главе 2. После формального описания встроенных типов давайте более подробно проанализируем литералы.

Целочисленные литералы

Целые числа, вероятно, следует считать наиболее часто используемым типом в рядовой программе. Любое целочисленное значение является целочисленным литералом. Примерами могут служить 1, 2, 3 и 42. Все они представляют собой десятичные значения, т.е. описывают числа по основанию 10. В целочисленных литералах допускается применять еще два вида чисел — *восьмеричные* (по основанию 8) и *шестнадцатеричные* (по основанию 16). Восьмеричные значения обозначаются в Java ведущим нулем. Нормальные десятичные числа не могут содержать ведущий ноль. Таким образом, выглядящее допустимым значение 09 вызовет ошибку на этапе компиляции, поскольку 9 находится за пределами восьмеричного диапазона от 0 до 7. Программисты чаще используют для чисел основание 16, которое точно соответствует размерам слов по модулю 8, таким как 8, 16, 32 и 64 бита. Шестнадцатеричная константа указывается с ведущим нулем и буквой "X" (`0x` или `0X`). Диапазон шестнадцатеричных цифр — от 0 до 15, так что числа от 10 до 15 заменяются буквами от `A` до `F` (или от `a` до `f`).

Целочисленные литералы создают значения типа `int`, которые в Java являются 32-битными целыми числами. Учитывая тот факт, что язык Java строго типизирован, вам может быть интересно, как присвоить целочисленный литерал одному из других целочисленных типов Java, скажем, `byte` или `long`, не вызывая ошибки несоответствия типов. К счастью, такие ситуации легко разрешаются. Когда литеральное значение присваивается переменной `byte` или `short`, ошибка не генерируется, если литеральное значение находится

в пределах диапазона допустимых значений целевого типа. Целочисленный литерал всегда можно присвоить переменной `long`. Тем не менее, для указания литерала `long` понадобится явно сообщить компилятору, что литеральное значение имеет тип `long`. Это делается путем добавления к литералу буквы "L" в верхнем или нижнем регистре. Например, `0x7fffffffffffffL` или `9223372036854775807L` — наибольшее значение типа `long`. Целое число можно присваивать переменной `char`, если оно находится в пределах допустимого диапазона.

Указывать целочисленные литералы можно также в двоичной форме, добавляя к значению префикс `0b` или `0B`. Скажем, вот как задать десятичное значение 10 с помощью двоичного литерала:

```
int x = 0b1010;
```

Помимо прочего, добавление двоичных литералов упрощает ввод значений, применяемых в качестве битовых масок. Десятичное (или шестнадцатеричное) представление значения визуально не передает смысл битовой маски, а двоичный литерал его передает.

В целочисленном литерале можно указывать один или несколько символов подчеркивания, которые упрощают чтение больших целочисленных литералов. При компиляции литерала символы подчеркивания отбрасываются. Например, в следующей строке:

```
int x = 123_456_789;
```

переменной x будет присвоено значение `123456789`. Символы подчеркивания игнорируются. Они могут использоваться только для разделения цифр. Символы подчеркивания не могут находиться в начале или в конце литерала. Однако между цифрами разрешено применять более одного символа подчеркивания. Например, показанный ниже код совершенно допустим:

```
int x = 123___456___789;
```

Использовать символы подчеркивания в целочисленных литералах особенно удобно при кодировании таких элементов, как телефонные номера, идентификационные номера заказчиков, номера деталей и т.д. Они также полезны для визуального группирования при указании двоичных литералов. Скажем, двоичные значения часто визуально группируются в блоки по четыре цифры:

```
int x = 0b1101_0101_0001_1010;
```

Литералы с плавающей точкой

Числа с плавающей точкой представляют десятичные значения с дробной частью. Они могут быть выражены с применением либо стандартной, либо научной (экспоненциальной) формы записи. *Стандартная форма записи* образована из целой части числа, десятичной точки и дробной части. Например, `2.0`, `3.14159` и `0.6667` представляют допустимые числа с плавающей точкой в стандартной форме записи. В *научной форме записи* используется стандарт-

ная форма записи числа с плавающей точкой плюс суффикс, указывающий степень 10, на которую число должно быть умножено. Показатель степени обозначается буквой `E` или `e`, за которой следует положительное или отрицательное десятичное число. Примерами могут служить `6.022E23`, `314159E-05` и `2e+100`.

Литералы с плавающей точкой в Java по умолчанию имеют тип `double`. Для указания литерала типа `float` к константе необходимо добавить букву `F` или `f`. Можно также явно указывать литерал `double`, добавляя букву `D` или `d`, хотя поступать так излишне. Назначаемый по умолчанию тип `double` занимает 64 бита памяти, а меньший тип `float` требует только 32 бита.

Шестнадцатеричные литералы с плавающей точкой тоже поддерживаются, но применяются редко. Они должны записываться в форме, похожей на научную, но вместо `E` или `e` используется буква `P` или `p`. Например, `0x12.2P2` является допустимым литералом с плавающей точкой. Значение после `P`, называемое *двоичным показателем степени*, указывает степень двойки, на которую умножается число. Следовательно, `0x12.2P2` представляет `72.5`.

В литералы с плавающей точкой можно встраивать один или несколько символов подчеркивания, что работает точно так же, как и для описанных ранее целочисленных литералов. Цель — облегчить чтение больших литералов с плавающей точкой. При компиляции литерала символы подчеркивания отбрасываются. Скажем, в следующей строке:

```
double num = 9_423_497_862.0;
```

переменной `num` будет присвоено значение `9423497862.0`. Символы подчеркивания игнорируются. Как и в случае с целочисленными литералами, символы подчеркивания могут применяться только для разделения цифр. Они не могут находиться в начале или в конце литерала. Однако между двумя цифрами разрешено указывать более одного символа подчеркивания. Кроме того, символы подчеркивания можно использовать в дробной части числа. Например, показанная ниже строка совершенно допустима:

```
double num = 9_423_497.1_0_9;
```

В данном случае дробной частью будет `.109`.

Булевские литералы

Булевские литералы просты. Значение типа `boolean` может иметь только два логических значения: `true` и `false`. Значения `true` и `false` не преобразуются в какое-то числовое представление. Литерал `true` в Java не равен 1, а литерал `false` не равен 0. В Java логические литералы можно присваивать только переменным, объявленным как `boolean`, или применять в выражениях с булевскими операциями.

Символьные литералы

Символы в Java являются индексами в наборе символов Unicode. Они представляют собой 16-битные значения, которые можно преобразовывать

в целые числа и обрабатывать с помощью целочисленных операций, таких как сложение и вычитание. Символьные литералы задаются внутри пары одинарных кавычек. Все видимые символы ASCII можно вводить прямо внутри кавычек, скажем, `'a'`, `'z'` и `'@'`. Для символов, которые ввести напрямую невозможно, существует несколько управляющих последовательностей, позволяющих ввести нужный символ, например, `'\''` для самого символа одинарной кавычки и `'\n'` для символа новой строки. Существует также механизм прямого ввода значения символа в восьмеричной или шестнадцатеричной форме. Для восьмеричной формы записи используется обратная косая черта, за которой следует трехзначное число. Скажем, `'\141'` — это буква `'a'`. Для шестнадцатеричной формы записи применяется обратная косая черта и буква u (\u), а за ними в точности четыре шестнадцатеричных цифры. Например, `'\u0061'` — это буква `'a'` из набора ISO-Latin-1, потому что старший байт равен нулю. `'\ua432'` — это символ японской катаканы. Управляющие последовательности символов перечислены в табл. 3.3.

Таблица 3.3. Символьные управляющие последовательности

Управляющая последовательность	Описание
\ddd	Восьмеричный символ (ddd)
\uxxxx	Шестнадцатеричный символ Unicode (xxxx)
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратная косая черта
\r	Возврат каретки
\n	Новая строка (также известная как перевод строки)
\f	Подача страницы
\t	Табуляция
\b	Забой
\s	Пробел (последовательность добавлена в JDK 15)
\конец-строки	Строка продолжения (применяется только к текстовым блокам; последовательность добавлена в JDK 15)

Строковые литералы

Строковые литералы в Java указываются таким же образом, как в большинстве других языков — путем заключения последовательности символов в пару двойных кавычек. Вот примеры строковых литералов:

```
"Hello World"
"two\nlines"
" \"This is in quotes\""
```

Управляющие последовательности и восьмеричная/шестнадцатеричная формы записи, которые были определены для символьных литералов, внутри строковых литералов работают точно так же. В отношении строковых литералов Java важно помнить, что они должны начинаться и заканчиваться в той же строчке, даже если она переносится. Для строковых литералов нет управляющей последовательности продолжения строки, как в ряде других языков. (Полезно отметить, что в версии JDK 15 в Java появилось функциональное средство, называемое текстовым блоком, которое обеспечивает больший контроль и гибкость, когда требуется несколько строчек текста. См. главу 17.)

На заметку! Как вам может быть известно, в некоторых других языках строки реализуются в виде массивов символов. Тем не менее, в Java ситуация иная. Строки на самом деле являются объектными типами. Как вы увидите далее в книге, по причине реализации строк как объектов Java обладает обширными возможностями обработки строк, которые характеризуются мощью и простотой использования.

Переменные

Переменная служит базовой единицей хранения в программе на Java. Переменная определяется комбинацией идентификатора, типа и необязательного инициализатора. Кроме того, все переменные имеют область видимости, определяющую их доступность, и время жизни. Эти элементы рассматриваются далее в главе.

Объявление переменной

В Java все переменные должны быть объявлены до того, как их можно будет использовать. Ниже показана основная форма объявления переменной:

```
тип идентификатор [= значение ][, идентификатор [= значение ] ...];
```

Здесь `тип` — один из примитивных типов Java либо имя класса или интерфейса. (Типы классов и интерфейсов обсуждаются позже в части I книги.) `Идентификатор` — это имя переменной. Вы можете инициализировать переменную, указав знак равенства и значение. Имейте в виду, что результатом выражения инициализации должно быть значение того же (или совместимого) типа, что и тип, указанный для переменной. Для объявления более одной переменной заданного типа применяется список, разделенный запятыми.

Взгляните на несколько примеров объявлений переменных различных типов. Обратите внимание, что некоторые из них включают инициализацию.

```
int a, b, c;              // объявить три переменных типа int, a, b и c
int d = 3, e, f = 5;      // объявить еще три переменных
                          //   типа int с инициализацией d и f
byte z = 22;              // инициализировать z
double pi = 3.14159;      // объявить приближенное значение pi
char x = 'x';             // переменная x имеет значение 'x'
```

Выбранные вами идентификаторы не имеют в своих именах ничего, что указывало бы на их тип. Java позволяет любому корректно сформированному идентификатору иметь любой объявленный тип.

Динамическая инициализация

Хотя в предыдущих примерах в качестве инициализаторов использовались только константы, Java позволяет инициализировать переменные динамически с применением любого выражения, действительного на момент объявления переменной.

Например, вот короткая программа, которая вычисляет длину гипотенузы прямоугольного треугольника при известных длинах его катетов:

```
// Демонстрация использования динамической инициализации.
class DynInit {
  public static void main(String[] args) {
    double a = 3.0, b = 4.0;

    // Переменная c инициализируется динамически.
    double c = Math.sqrt(a * a + b * b);

    System.out.println("Длина гипотенузы равна " + c);
  }
}
```

Здесь объявляются три локальные переменные — `a`, `b` и `c`. Первые две, `a` и `b`, инициализируются константами. Однако переменная `c` инициализируется динамически длиной гипотенузы (по теореме Пифагора). В программе используется еще один встроенный метод Java, `sqrt()`, являющийся членом класса `Math`, который вычисляет квадратный корень своего аргумента. Ключевой момент в том, что выражение инициализации может содержать любой элемент, действительный во время инициализации, в том числе вызовы методов, другие переменные или же литералы.

Область видимости и время жизни переменных

До сих пор все применяемые переменные были объявлены в начале метода `main()`. Тем не менее, Java позволяет объявлять переменные внутри любого блока. Как объяснялось в главе 2, блок начинается с открывающей фигурной скобки и заканчивается закрывающей фигурной скобкой. Блок определяет *область видимости*. Таким образом, каждый раз, начиная новый блок, вы создаете новую область видимости. Область видимости устанавливает, какие объекты видны другим частям вашей программы. Она также определяет время жизни этих объектов.

Нередко мыслят в терминах двух категорий областей видимости: глобальной и локальной. Однако такие традиционные области видимости плохо вписываются в строгую объектно-ориентированную модель Java. Хотя вполне возможно создать глобальную область видимости, это скорее исключение, нежели правило. Две основные области видимости в Java определяются классом и методом. Даже такое различие несколько искусственно. Тем не менее, поскольку область видимости класса обладает несколькими уникальными свойствами и атрибутами, которые не применяются к области видимости, определенной методом, то различие обретает смысл. Из-за отличий обсуждение области видимости класса (и объявленных в ней переменных) отложено

до главы 6, где будут описаны классы. Сейчас мы исследуем только области видимости, определяемые методом или внутри него.

Область видимости, определяемая методом, начинается с его открывающей фигурной скобки. Однако если у метода есть параметры, то они тоже входят в область видимости метода. Область видимости метода заканчивается закрывающей фигурной скобкой. Такой блок кода называется *телом метода*.

Как правило, переменные, объявленные внутри области видимости, не будут доступны в коде за рамками этой области. Таким образом, когда вы объявляете переменную в области видимости, то локализуете ее и защищаете от несанкционированного доступа и/или модификации. Действительно, правила области видимости обеспечивают основу для инкапсуляции. Переменная, объявленная внутри блока, называется *локальной переменной*.

Области могут быть вложенными. Например, создавая блок кода, вы создаете новую вложенную область. В таком случае внешняя область видимости охватывает внутреннюю область видимости. В итоге объекты, объявленные во внешней области, будут видимыми коду во внутренней области. Тем не менее, обратное утверждение неверно. Объекты, объявленные во внутренней области, не будут видны за ее пределами.

Чтобы лучше понять влияние вложенных областей видимости, взгляните на следующую программу:

```
// Демонстрация области видимости блока кода.
class Scope {
  public static void main(String[] args) {
    int x;          // переменная известна всему коду внутри main()

    x = 10;
    if(x == 10) {   // начало новой области видимости
      int y = 20;   // переменная известна только этому блоку

      // x и y здесь известны.
      System.out.println("x и y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100;     // Ошибка! Переменная y здесь неизвестна.

    // Переменная x здесь по-прежнему известна.
    System.out.println("Значение x равно " + x);
  }
}
```

Как указано в комментариях, переменная x объявлена в начале области видимости метода `main()` и доступна всему последующему коду внутри `main()`. Переменная y объявлена внутри блока `if`. Поскольку блок определяет область видимости, y видна только остальному коду внутри этого блока. Вот почему строка `y = 100;` вне блока `if` закомментирована. Если удалить ведущие символы комментария, тогда возникнет ошибка на этапе компиляции, потому что переменная y не видна за пределами своего блока. Внутри блока `if` можно использовать переменную x, т.к. код внутри блока (т.е. во вложенной области видимости) имеет доступ к переменным, объявленным в объемлющей области.

Внутри блока переменные могут быть объявлены в любой момент, но действительны только после их объявления. Таким образом, если вы определяете переменную в начале метода, то она доступна всему коду внутри этого метода. И наоборот, если вы объявляете переменную в конце блока, то она по существу бесполезна, потому что код не будет иметь к ней доступа. Например, следующий фрагмент кода приведет к ошибке, т.к. переменную count нельзя использовать до ее объявления:

```
// Здесь присутствует ошибка!
count = 100;  // Переменную count нельзя использовать до ее объявления!
int count;
```

Необходимо запомнить еще один важный момент: переменные создаются при входе в их область видимости и уничтожаются при выходе из их области видимости. Другими словами, переменная не будет хранить свое значение после того, как покинет пределы области видимости. Следовательно, переменные, объявленные в методе, не сохраняют свои значения между вызовами этого метода. Кроме того, переменная, объявленная внутри блока, утратит свое значение при выходе из блока. Таким образом, время жизни переменной ограничено ее областью видимости.

Если объявление переменной содержит инициализатор, тогда переменная будет повторно инициализироваться при каждом входе в блок, где она объявлена. Например, рассмотрим показанную ниже программу:

```
// Демонстрация времени жизни переменной.
class LifeTime {
  public static void main(String[] args) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1;  // переменная y инициализируется при каждом входе в блок
      System.out.println("Значение y равно " + y);  // всегда выводится -1
      y = 100;
      System.out.println("Теперь значение y равно " + y);
    }
  }
}
```

Программа генерирует следующий вывод:

```
Значение y равно -1
Теперь значение y равно 100
Значение y равно -1
Теперь значение y равно 100
Значение y равно -1
Теперь значение y равно 100
```

Как видите, переменная y повторно инициализируется значением -1 при каждом входе во внутренний цикл for. Несмотря на то что впоследствии ей присваивается значение 100, это значение утрачивается.

Последнее замечание: хотя блоки могут быть вложенными, вы не можете объявить переменную с тем же именем, что и у переменной во внешней области видимости. Скажем, показанная далее программа некорректна:

```
// Эта программа не скомпилируется.
class ScopeErr {
  public static void main(String[] args) {
    int bar = 1;
    {                  // создать новую область видимости
      int bar = 2;     // ошибка на этапе компиляции - переменная bar
                       //   уже определена
    }
  }
}
```

Преобразование и приведение типов

Если у вас есть опыт программирования, то вы уже знаете, что довольно часто значение одного типа присваивается переменной другого типа. В случае, когда два типа совместимы, компилятор Java автоматически выполнит преобразование. Скажем, значение типа `int` всегда можно присваивать переменной типа `long`. Однако не все типы совместимы и потому не все преобразования типов неявно разрешены. Например, автоматическое преобразование из `double` в `byte` не определено. К счастью, обеспечить преобразование между несовместимыми типами все-таки можно. Для этого придется применять *приведение*, которое выполняет явное преобразование между несовместимыми типами. Давайте более подробно рассмотрим автоматические преобразования и приведение типов.

Автоматические преобразования в Java

Когда значение одного типа присваивается переменной другого типа, автоматическое преобразование типов происходит в случае удовлетворения следующих двух условий:

- два типа совместимы;
- целевой тип больше исходного типа.

При соблюдении указанных двух условий выполняется *расширяющее преобразование*. Например, тип `int` всегда достаточно велик, чтобы хранить все допустимые значения `byte`, поэтому явное приведение не требуется.

С точки зрения расширяющих преобразований числовые типы, включая целочисленные типы и типы с плавающей точкой, совместимы друг с другом. Тем не менее, автоматические преобразования числовых типов в `char` или `boolean` не предусмотрены. Кроме того, типы `char` и `boolean` не совместимы друг с другом.

Как упоминалось ранее, компилятор Java также выполняет автоматическое преобразование типов при сохранении литеральной целочисленной константы в переменные типа `byte`, `short`, `long` или `char`.

Приведение несовместимых типов

Несмотря на удобство автоматического преобразования типов, оно не сможет удовлетворить все требования. Скажем, что делать, когда нужно при-

своить значение типа int переменной типа byte? Такое преобразование не будет выполняться автоматически, потому что тип byte меньше типа int. Преобразование подобного рода иногда называют *сужающим преобразованием*, поскольку значение явно сужается, чтобы уместиться в целевой тип.

Преобразование между двумя несовместимыми типами создается с использованием приведения. *Приведение* представляет собой просто явное преобразование типа и имеет следующую общую форму:

```
(целевой-тип) значение
```

Здесь целевой-тип указывает желаемый тип, в который необходимо преобразовать заданное значение. Например, в показанном далее фрагменте кода тип int приводится к byte. Если целочисленное значение выходит за пределы диапазона типа byte, тогда оно уменьшается по модулю (остатку от целочисленного деления) диапазона byte.

```
int a;
byte b;
// ...
b = (byte) a;
```

Когда переменной целочисленного типа присваивается значение с плавающей точкой, будет происходить другой тип преобразования: *усечение*. Как известно, целые числа не имеют дробных частей. Таким образом, в случае присваивания переменной целочисленного типа значения с плавающей точкой дробная часть теряется. Например, если целочисленной переменной присваивается значение 1.23, то результирующим значением оказывается просто 1, а часть 0.23 усекается. Разумеется, если размер целого компонента числа с плавающей точкой слишком велик, чтобы уместиться в целевой целочисленный тип, то значение будет уменьшено по модулю диапазона целевого типа.

В следующей программе демонстрируется несколько преобразований типов, требующих приведений:

```
// Демонстрация приведений.
class Conversion {
  public static void main(String[] args) {
    byte b;
    int i = 257;
    double d = 323.142;

    System.out.println("\nПреобразование int в byte.");
    b = (byte) i;
    System.out.println("i и b: " + i + " " + b);

    System.out.println("\nПреобразование double в int.");
    i = (int) d;
    System.out.println("d и i: " + d + " " + i);

    System.out.println("\nПреобразование double в byte.");
    b = (byte) d;
    System.out.println("d и b: " + d + " " + b);
  }
}
```

Вот вывод, генерируемый программой:

```
Преобразование int в byte.
i и b: 257 1

Преобразование double в int.
d и i: 323.142 323

Преобразование double в byte.
d и b: 323.142 67
```

Давайте обсудим каждое преобразование. Когда значение 257 приводится к типу byte, результатом будет остаток от деления 257 на 256 (диапазон byte), который в данном случае равен 1. Когда значение переменной d преобразуется в тип int, его дробная часть утрачивается. Когда значение переменной d преобразуется в тип byte, его дробная часть теряется, а значение уменьшается по модулю 256, что в этом случае дает 67.

Автоматическое повышение типов в выражениях

Помимо присваивания есть еще одно место, где могут происходить определенные преобразования типов: выражения. Давайте выясним причину. Точность, требуемая для представления промежуточного значения в выражении, иногда превышает диапазон допустимых значений типа любого из операндов. Например, возьмем показанное ниже выражение:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

Результат промежуточного члена a * b может легко выйти за пределы диапазона любого из его операндов типа byte. Чтобы решить проблему такого рода, при вычислении выражения каждый операнд типа byte, short или char автоматически повышается до int. Это означает, что подвыражение a * b вычисляется с применением целочисленных, а не байтовых значений. Таким образом, значение 2000, т.е. результат вычисления промежуточного выражения 50 * 40, будет допустимым, несмотря на то, что a и b объявлены с типом byte.

Каким бы полезным ни было автоматическое повышение, оно может вызывать сбивающие с толку ошибки на этапе компиляции. Скажем, следующий с виду правильный код вызывает проблему:

```
byte b = 50;
b = b * 2;  // Ошибка! Нельзя присваивать значение int переменной byte!
```

В коде предпринимается попытка сохранить 50 * 2, т.е. совершенно допустимое значение byte, обратно в переменную типа byte. Но поскольку при вычислении выражения операнды автоматически повышаются до int, результат тоже повышается до int. Таким образом, результат выражения теперь имеет тип int, который нельзя присвоить переменной типа byte, не используя приведение. Это справедливо даже в том случае, если присваиваемое значение умещается в целевой тип, как в примере выше.

В ситуациях, когда вам понятны последствия переполнения, вы должны применять явное приведение, например, как в выражении ниже, которое выдает корректное значение 100:

```
byte b = 50;
b = (byte)(b * 2);
```

Правила повышения типов

В Java определено несколько *правил повышения типов*, которые применяются к выражениям. Вот как они выглядят. Первым делом все значения byte, short и char повышаются до int, как только что было описано. Если один операнд имеет тип long, то все выражение повышается до long. Если один операнд имеет тип float, тогда все выражение повышается до float. Если какой-либо из операндов имеет тип double, то результат будет иметь тип double.

В представленной далее программе демонстрируется повышение каждого значения в выражении для соответствия типу второго операнда в каждой двоичной операции:

```
class Promote {
  public static void main(String[] args) { byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
    double result = (f * b) + (i / c) - (d * s);
    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
    System.out.println("result = " + result);
  }
}
```

Давайте более внимательно рассмотрим повышения типов, которые происходят в следующей строке программы:

```
double result = (f * b) + (i / c) - (d * s);
```

В первом подвыражении f * b, тип переменной b повышается float и результатом подвыражения будет значение float. В подвыражении i / c тип переменной c повышается до int и результатом подвыражения будет значение int. В подвыражении d * s тип переменной s повышается до double и результатом подвыражения будет значение double. Наконец, принимаются во внимание три промежуточных значения: float, int и double. Результат сложения значений float и int имеет тип float. Тип результата вычитания последнего промежуточного значения double из результирующего значения float повышается до double, который и будет типом окончательного результата выражения.

Массивы

Массив — это группа переменных одного типа, к которой можно обращаться по общему имени. Можно создавать массивы любого типа с одним или большим количеством измерений. Доступ к определенному элементу массива осуществляется по его индексу. Массивы предлагают удобные средства группирования связанной информации.

Одномерные массивы

Одномерный массив по существу представляет собой список переменных одного типа. Чтобы создать массив, сначала необходимо создать переменную массива желаемого типа. Вот общая форма объявления одномерного массива:

```
тип[] имя-переменной;
```

Здесь тип объявляет тип элементов массива, который называется также базовым типом. Тип элементов определяет тип данных каждого элемента, содержащегося в массиве. Таким образом, тип элементов массива определяет, данные какого типа будут храниться в массиве. Например, в следующем примере объявляется массив по имени month_days с типом "массив целых чисел":

```
int[] month_days;
```

Хотя такое объявление устанавливает тот факт, что month_days является переменной типа массива, на самом деле никакого массива не существует. Чтобы связать month_days с фактическим физическим массивом целых чисел, потребуется разместить его в памяти с помощью new и назначить month_days. Как будет объясняться, new — это специальная операция, которая выделяет память.

Операция new более подробно рассматривается в следующей главе, но ее нужно использовать сейчас, чтобы размещать в памяти массивы. Общая форма операции new применительно к одномерным массивам выглядит так:

```
переменная-типа-массива = new тип [размер];
```

Здесь тип указывает тип размещаемых в памяти данных, размер устанавливает количество элементов в массиве, а переменная-типа-массива представляет собой переменную, связанную с массивом. То есть, чтобы использовать new для размещения массива, вы должны указать тип и количество элементов в массиве. Элементы в массиве, размещенном в памяти операцией new, будут автоматически инициализированы нулем (для числовых типов), значением false (для булевского типа) или значением null (для ссылочных типов, описанных в следующей главе). В показанном ниже примере размещается 12-элементный массив целых чисел и связывается с переменной month_days:

```
month_days = new int[12];
```

После выполнения этого оператора month_days будет ссылаться на массив из 12 целых чисел. Вдобавок все элементы в массиве будут инициализированы нулем.

Имеет смысл повторить: получение массива — двухэтапный процесс. Во-первых, вы обязаны объявить переменную нужного типа массива. Во-вторых, вы должны выделить память, в которой будет храниться массив, с применением операции new и назначить ее переменной типа массива. Таким образом, в Java все массивы размещаются в памяти динамически. Если концепция динамического размещения вам незнакома, не переживайте. Она будет подробно описана далее в книге.

После размещения массива вы можете получать доступ к определенному элементу массива, указывая его индекс в квадратных скобках. Индексы массивов начинаются с нуля. Например, следующий оператор присваивает значение 28 второму элементу массива month_days:

```
month_days[1] = 28;
```

Представленная далее строка отображает значение, хранящееся по индексу 3:

```
System.out.println(month_days[3]);
```

Собирая воедино все части, можно написать программу, которая создает массив с количеством дней в каждом месяце:

```
// Демонстрация использования одномерного массива.
class Array {
  public static void main(String[] args) {
    int[] month_days;
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[7] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
    month_days[10] = 30;
    month_days[11] = 31;
    System.out.println("В апреле " + month_days[3] + " дней.");
  }
}
```

В результате запуска программа выведет количество дней в апреле. Как уже упоминалось, индексы массивов Java начинаются с нуля, так что количество дней в апреле хранится в элементе month_days[3] и составляет 30.

Разрешено объединять объявление переменной типа массива с выделением для него памяти:

```
int[] month_days = new int[12];
```

Именно так обычно поступают в профессионально написанных программах на Java.

Массивы могут инициализироваться при их объявлении. Процесс почти такой же, как при инициализации простых типов. *Инициализатор массива* представляет собой список разделенных запятыми выражений, заключенный в фигурные скобки. Запятые разделяют значения элементов массива. Массив будет автоматически создаваться достаточно большим, чтобы вместить количество элементов, указанное в инициализаторе массива. Нет необходимости использовать new. Например, для хранения количества дней в месяцах в следующем коде создается инициализированный массив целых чисел:

```
// Улучшенная версия предыдущей программы.
class AutoArray {
  public static void main(String[] args) {
    int[] month_days = { 31, 28, 31, 30, 31, 30,
                         31, 31, 30, 31, 30, 31 };
    System.out.println("В апреле " + month_days[3] + " дней.");
  }
}
```

В результате запуска программы вы увидите такой же вывод, как и у предыдущей версии.

Исполняющая среда Java строго проверяет, не попытались ли вы случайно сохранить или сослаться на значения за пределами диапазона массива. Она проверит, что все индексы массива находятся в правильном диапазоне. Например, исполняющая среда проверит значение каждого индекса в month_days, чтобы удостовериться, что оно находится внутри диапазона от 0 до 11 включительно. Если вы попытаетесь получить доступ к элементам за пределами диапазона массива (отрицательные числа или числа, превышающие длину массива), то получите ошибку времени выполнения.

Вот еще один пример, в котором применяется одномерный массив. В нем вычисляется среднее для множества чисел.

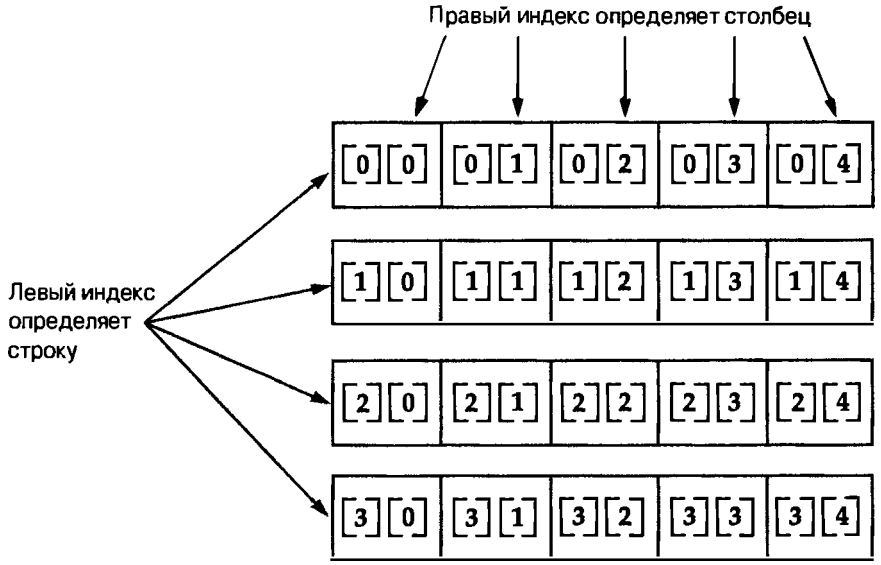
```
// Вычисление среднего для массива значений.
class Average {
  public static void main(String[] args) {
    double[] nums = {10.1, 11.2, 12.3, 13.4, 14.5};
    double result = 0; int i;

    for(i=0; i<5; i++)
      result = result + nums[i];
    System.out.println("Среднее значение: " + result / 5);
  }
}
```

Многомерные массивы

Многомерные массивы в Java реализованы как массивы массивов. Чтобы объявить переменную многомерного массива, указывайте каждый дополнительный индекс, используя еще один набор квадратных скобок. Скажем, в следующем примере объявляется переменная типа двумерного массива по имени twoD:

```
int[][] twoD = new int[4][5];
```

Здесь размещается память для массива 4×5 и назначается переменной twoD. Внутренне матрица реализована в виде *массива массивов* значений int. Концептуально такой массив будет выглядеть, как показано на рис. 3.1.



Дано: `int[][] twoD = new int[4][5];`

Рис. 3.1. Концептуальное представление двумерного массива 4×5

В приведенной ниже программе элемент массива нумеруются слева направо и сверху вниз, после чего отображаются их значения:

```
// Демонстрация использования многомерного массива.
class TwoDArray {
  public static void main(String[] args) {
    int[][] twoD= new int[4][5];
    int i, j, k = 0;
    for(i=0; i<4; i++) for(j=0; j<5; j++) {
      twoD[i][j] = k;
      k++;
    }
    for(i=0; i<4; i++) {
      for(j=0; j<5; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

Программа генерирует следующий вывод:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

При размещении многомерного массива необходимо указывать память только для первого (самого левого) измерения. Остальные измерения можно размещать по отдельности. Например, в показанном далее коде выделяется память для первого измерения массива twoD, когда он объявляется. Выделение памяти для второго измерения делается отдельно.

```
int[][] twoD = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

Хотя в такой ситуации индивидуальное размещение массивов второго измерения не дает каких-то преимуществ, в других случаях такие преимущества могут быть. Скажем, при выделении памяти под измерения по отдельности вам не нужно размещать одинаковое количество элементов для каждого измерения. Как утверждалось ранее, поскольку многомерные массивы на самом деле являются массивами массивов, длина каждого массива находится под вашим контролем. Например, в следующей программе создается двумерный массив, в котором размеры массивов во втором измерении не равны:

```
// Ручное размещение массивов разных размеров во втором изменении.
class TwoDAgain {
  public static void main(String[] args) {
    int[][] twoD = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<i+1; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

Программа генерирует показанный ниже вывод:

```
0
1 2
3 4 5
6 7 8 9
```

Созданный программой массив схематически изображен на рис. 3.2.

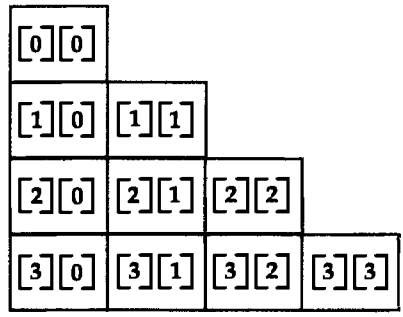


Рис. 3.2. Двумерный массив, в котором размеры массивов во втором измерении не равны

Ступенчатые (или нерегулярные) многомерные массивы могут оказаться неподходящими для многих приложений, потому что они противоречат тому, что люди ожидают найти при встрече с многомерным массивом. Однако в некоторых ситуациях нерегулярные массивы можно эффективно использовать. Скажем, если вам нужен очень большой разреженный двумерный массив (т.е. такой, где задействованы не все элементы), тогда нерегулярный массив может стать идеальным решением.

Многомерные массивы можно инициализировать. Для этого просто заключите инициализатор каждого измерения в собственный набор фигурных скобок. В следующей программе создается матрица, в которой каждый элемент содержит произведение индексов строки и столбца. Также обратите внимание, что внутри инициализаторов массивов можно применять выражения и литеральные значения.

```
// Инициализация двумерного массива.
class Matrix {
  public static void main(String[] args) {
    double[][] m = {
      { 0*0, 1*0, 2*0, 3*0 },
      { 0*1, 1*1, 2*1, 3*1 },
      { 0*2, 1*2, 2*2, 3*2 },
      { 0*3, 1*3, 2*3, 3*3 }
    };
    int i, j;
    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
      System.out.println();
    }
  }
}
```

Запустив программу, вы получите такой вывод:

```
0.0 0.0 0.0 0.0
0.0 1.0 2.0 3.0
0.0 2.0 4.0 6.0
0.0 3.0 6.0 9.0
```

Как видите, все строки в массиве инициализируются в соответствии со списками инициализации.

Давайте рассмотрим еще один пример использования многомерного массива. В показанной ниже программе создается трехмерный массив 3×4×5, затем в каждый элемент массива помещается произведение его индексов, после чего результирующие произведения отображаются.

```
// Демонстрация использования трехмерного массива.
class ThreeDMatrix {
  public static void main(String[] args) {
    int[][][] threeD = new int[3][4][5];
    int i, j, k;

    for(i=0; i<3; i++)
      for(j=0; j<4; j++)
        for(k=0; k<5; k++)
          threeD[i][j][k] = i * j * k;
    for(i=0; i<3; i++) {
      for(j=0; j<4; j++) {
        for(k=0; k<5; k++)
          System.out.print(threeD[i][j][k] + " ");
        System.out.println();
      }
      System.out.println();
    }
  }
}
```

Вот какой вывод генерирует программа:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

Альтернативный синтаксис объявления массивов

Существует вторая форма объявления массива:

тип имя-переменной[];

Здесь квадратные скобки расположены после имени переменной массива, а не после спецификатора типа. Например, следующие два объявления эквивалентны:

```
int a1[] = new int[3];
int[] a2 = new int[3];
```

Представленные далее объявления тоже эквивалентны:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

Эта альтернативная форма объявления обеспечивает удобство при преобразовании кода из C/C++ в Java. Кроме того, она позволяет объявлять в одном операторе объявления и переменные с типами массивов, и переменные с типами, отличающимися от массивов. В настоящее время альтернативная форма используется реже, но знать ее по-прежнему важно, поскольку в Java разрешены обе формы объявления массивов.

Знакомство с выведением типов локальных переменных

Не так давно в язык Java было добавлено новое функциональное средство, называемое *выведением типов локальных переменных*. Для начала давайте вспомним о двух важных аспектах, касающихся переменных. Во-первых, все переменные в Java должны быть объявлены до их использования. Во-вторых, переменная может быть инициализирована значением при ее объявлении. Кроме того, когда переменная инициализируется, тип инициализатора должен быть таким же, как объявленный тип переменной (или допускать преобразование в него). Таким образом, в принципе нет нужды указывать явный тип для инициализируемой переменной, потому что он может быть выведен по типу ее инициализатора. Конечно, в прошлом такое выведение не поддерживалось, и все переменные требовали явно объявленного типа вне зависимости от того, инициализировались они или нет. На сегодняшний день ситуация изменилась.

Начиная с JDK 10, можно позволить компилятору выводить тип локальной переменной на основе типа ее инициализатора, избегая явного указания типа. Выведение типов локальных переменных предлагает несколько преимуществ. Например, оно может упростить код, устраняя необходимость в избыточном указании типа переменной, когда тип может быть выведен из ее инициализатора. Выведение типов локальных переменных способно упрощать объявления в случаях, когда имя типа имеет довольно большую длину, как у некоторых имен классов. Оно также может быть полезно, когда тип трудно различить или его нельзя обозначить. (Примером типа, который не может быть обозначен, является тип анонимного класса, обсуждаемый в главе 25.) Более того, выведение типов локальных переменных стало обычной частью современной программной среды. Его включение в Java помогает поддерживать язык в актуальном состоянии с учетом меняющихся тенденций в проектировании языков. Для поддержки выведения типов локальных переменных было добавлено контекстно-чувствительное ключевое слово `var`.

Чтобы задействовать выведение типов локальных переменных, переменная должна быть объявлена с ключевым словом `var` в качестве имени типа и включать инициализатор.

Например, вот как вы объявляли бы локальную двойную переменную по имени avg, инициализируемую значением 10.0, в прошлом:

```
double avg = 10.0;
```

С применением выведения типа объявление переменной avg теперь можно записать так:

```
var avg = 10.0;
```

В обоих случаях переменная avg будет иметь тип double. В первом случае ее тип указывается явно, а во втором случае выводится как double, т.к. инициализатор 10.0 имеет тип double.

Как уже упоминалось, ключевое слово var является зависимым от контекста. Когда var используется в качестве имени типа в контексте объявления локальной переменной, оно сообщает компилятору о том, что тип объявляемой переменной должен выводиться на основе типа инициализатора. Таким образом, в объявлении локальной переменной ключевое слово var служит заполнителем фактически выведенного типа. Тем не менее, когда ключевое слово var применяется в большинстве других мест, оно будет просто определяемым пользователем идентификатором без особого смысла. Например, следующее объявление по-прежнему допустимо:

```
int var = 1;    // В этом случае var - просто определяемый
                // пользователем идентификатор.
```

В данном случае тип явно указан как int, а var представляет собой имя объявляемой переменной. Несмотря на зависимость от контекста, есть несколько мест, где ключевое слово var использовать не разрешено. Скажем, его нельзя применять в качестве имени класса.

Предшествующие обсуждения воплощены в следующей программе:

```
// Простая демонстрация выведения типов локальных переменных.
class VarDemo {
  public static void main(String[] args) {
    // Использовать выведение типов для определения типа переменной
    // по имени avg. В этом случае выводится тип double.
    var avg = 10.0;
    System.out.println("Значение avg: " + avg);
    // В следующем контексте var - не предопределенный идентификатор,
    // а просто определяемое пользователем имя переменной.
    int var = 1;
    System.out.println("Значение var: " + var);
    // Интересно отметить, что в следующем фрагменте кода var используется
    // и как тип объявления, и как имя переменной в инициализаторе.
    var k = -var;
    System.out.println("Значение k: " + k);
  }
}
```

Ниже показан вывод программы:

```
Значение avg: 10.0
Значение var: 1
Значение k: -1
```

В предыдущем примере ключевое слово var использовалось для объявления только простых переменных, но var можно применять также для объявления массивов. Например:

```
var myArray = new int[10];          // Допустимый код.
```

Обратите внимание, что ни var, ни myArray не имеют скобок. Взамен предполагается, что тип myArray выводится в int[]. Кроме того, использовать квадратные скобки в левой части объявления var нельзя. Таким образом, оба следующих объявления ошибочны:

```
var[] myArray = new int[10];        // Ошибка!
var myArray[] = new int[10];        // Ошибка!
```

В первой строке предпринимается попытка снабдить квадратными скобками ключевое слово var, а во второй — переменную myArray. В обоих случаях применять квадратные скобки некорректно, т.к. тип выводится из типа инициализатора.

Важно подчеркнуть, что var может использоваться для объявления переменной только тогда, когда эта переменная инициализирована. Например, показанный далее оператор ошибочен:

```
var counter;  // Ошибка! Требуется инициализатор.
```

Также помните о том, что ключевое слово var можно применять только для объявления локальных переменных. Его нельзя использовать, например, при объявлении переменных экземпляра, параметров или возвращаемых типов.

В то время как предыдущее обсуждение и примеры позволили вам ознакомиться с основами выведения типов локальных переменных, они не продемонстрировали его полную мощь. Как вы увидите в главе 7, выведение типов локальных переменных особенно эффективно для сокращения объявлений, содержащих длинные имена классов. Его также можно задействовать при работе с обобщенными типами (см. главу 14), оператором try с ресурсами (см. главу 13) и циклом for (см. главу 5).

Некоторые ограничения var

В дополнение к ограничениям, которые упоминались в предыдущем обсуждении, с применением var связано несколько других ограничений. Можно объявлять только одну переменную за раз, для переменной нельзя использовать null в качестве инициализатора и объявляемая переменная не может присутствовать в выражении инициализатора. Хотя с применением var можно объявить тип массива, ключевое слово var нельзя использовать с инициализатором массива. Например, следующий оператор допустим:

```
var myArray = new int[10];          // Допустим
```

но показанный ниже оператор — нет:

```
var myArray = { 1, 2, 3 };          // Ошибочен
```

Как отмечалось ранее, ключевое слово var не разрешено применять для имени класса. Его также не допускается использовать в качестве имени дру-

гих ссылочных типов, включая интерфейс, перечисление или аннотацию, либо в качестве имени параметра обобщенного типа, что рассматривается далее в книге. Существуют еще два ограничения, которые относятся к функциональным средствам Java, описанным в последующих главах, но упомянутым здесь для полноты картины. Выведение типов локальных переменных нельзя применять для объявления типа исключения, перехваченного оператором `catch`. Кроме того, ни лямбда-выражения, ни ссылки на методы не разрешено использовать в качестве инициализаторов.

На заметку! На момент написания книги некоторые читатели будут иметь дело со средами Java, не поддерживающими выведение типов локальных переменных. Чтобы все читатели книги могли компилировать и запускать как можно больше примеров кода, в большинстве программ в оставшейся части этого издания книги выведение типов локальных переменных применяться не будет. Использование полного синтаксиса объявления также позволяет сразу понять, переменная какого типа создается, что важно для самого примера кода. Разумеется, со временем вы должны обдумать применение в своем коде выведения типов локальных переменных.

Несколько слов о строках

Как вы могли заметить, в предыдущем обсуждении типов данных и массивов не упоминались строки или строковый тип данных. Причина вовсе не в том, что язык Java не поддерживает такой тип — он поддерживает. Просто строковый тип Java, называемый `String`, не является примитивным типом, равно как и обычным массивом символов. Наоборот, тип `String` определяет объект и потому его полное описание требует понимания нескольких характеристик, связанных с объектом. В итоге он будет рассматриваться в этой книге позже, после описания объектов. Однако чтобы вы могли использовать простые строки в примерах программ, необходимо следующее краткое введение.

Тип `String` предназначен для объявления строковых переменных. Можно также объявлять массивы строк. Строковой переменной может быть присвоена строковая константа в кавычках. Переменная типа `String` может быть присвоена другой переменной типа `String`. Объект типа `String` можно применять в качестве аргумента функции `println()`. Например, взгляните на следующий фрагмент кода:

```
String str = "Тестовая строка";
System.out.println(str);
```

Здесь `str` — это объект типа `String`. Ему присваивается строка `"Тестовая строка"`, которая и отображается оператором `println()`.

Позже вы увидите, что объекты `String` обладают многими особенностями и характеристиками, которые делают их довольно мощными и легкими в использовании. Тем не менее, в следующих нескольких главах вы будете применять их только в самой простой форме.

ГЛАВА

4 Операции

Язык Java предоставляет богатую среду операций. Большинство операций можно разделить на четыре группы: арифметические операции, побитовые операции, операции отношения и логические операции. В Java также определены дополнительные операции, обрабатывающие определенные особые ситуации. В этой главе описаны все операции Java кроме операции сравнения типов `instanceof`, исследуемой в главе 13, и операции стрелки (`->`), описанной в главе 15.

Арифметические операции

Арифметические операции используются в математических выражениях аналогично тому, как они применяются в алгебре, и перечислены в табл. 4.1.

Таблица 4.1. Арифметические операции языка Java

Операция	Описание
+	Сложение (также унарный плюс)
-	Вычитание (также унарный минус)
*	Умножение
/	Деление
%	Деление по модулю
++	Инкремент
+=	Сложение с присваиванием
-=	Вычитание с присваиванием
*=	Умножение с присваиванием
/=	Деление с присваиванием
%=	Деление по модулю с присваиванием
--	Декремент

Операнды арифметических операций должны иметь числовой тип. Арифметические операции нельзя использовать с типом boolean, но можно с типом char, поскольку в Java тип char по существу является подмножеством типа int.

Основные арифметические операции

Основные арифметические операции — сложение, вычитание, умножение и деление — ведут себя так, как и следовало ожидать для всех числовых типов. Унарный минус инвертирует свой единственный операнд. Унарный плюс просто возвращает значение своего операнда. Помните, что когда операция деления применяется к целочисленному типу, дробная часть к результату не присоединяется.

В следующей простой программе демонстрируется работа арифметических операций. В ней также иллюстрируется отличие между делением с плавающей точкой и целочисленным делением.

```
// Демонстрация основных арифметических операций.
class BasicMath {
  public static void main(String[] args) {
    // Арифметические операции со значениями int.
    System.out.println("Целочисленная арифметика");
    int a = 1 + 1;
    int b = a * 3;
    int c = b / 4;
    int d = c - a;
    int e = -d;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
    System.out.println("e = " + e);

    // Арифметические операции со значениями double.
    System.out.println("\nАрифметика с плавающей точкой");
    double da = 1 + 1;
    double db = da * 3;
    double dc = db / 4;
    double dd = dc - a;
    double de = -dd;
    System.out.println("da = " + da);
    System.out.println("db = " + db);
    System.out.println("dc = " + dc);
    System.out.println("dd = " + dd);
    System.out.println("de = " + de);
  }
}
```

Запустив программу, вы получите показанный ниже вывод:

```
Целочисленная арифметика
a = 2
b = 6
```

```
c = 1
d = -1
e = 1

Арифметика с плавающей точкой
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

Операция деления по модулю

Операция деления по модулю, %, возвращает остаток от деления. Ее можно применять к типам с плавающей точкой, а также к целочисленным типам. В следующей программе демонстрируется использование операции %:

```
// Демонстрация работы операции %.
class Modulus {
  public static void main(String[] args) {
    int x = 42;
    double y = 42.25;

    System.out.println("x по модулю 10 = " + x % 10);
    System.out.println("y по модулю 10 = " + y % 10);
  }
}
```

В результате запуска программы вы получите такой вывод:

```
x по модулю 10 = 2
y по модулю 10 = 2.25
```

Составные арифметические операции присваивания

Язык Java предоставляет специальные операции, объединяющие арифметические операции с присваиванием. Вероятно, вам известно, что в программах довольно часто встречаются операторы следующего вида:

```
a = a + 4;
```

В Java показанный оператор можно переписать следующим образом:

```
a += 4;
```

В этой версии оператора применяется *составная операция присваивания* +=. Оба оператора выполняют одно и то же действие: увеличивают значение a на 4.

Вот еще один пример:

```
a = a % 2;
```

который можно выразить так:

```
a %= 2;
```

В данном случае операция %= получает остаток от деления a/2 и помещает результат обратно в a.

Составные операции присваивания предусмотрены для всех арифметических бинарных операций. Таким образом, любой оператор вида:

```
переменная = переменная операция выражение;
```

может быть переписан следующим образом:

```
переменная операция= выражение;
```

Составные операции присваивания обеспечивают два преимущества. Во-первых, они уменьшают объем набираемого кода, потому что являются "сокращенной формой" для своих длинных эквивалентов. Во-вторых, в некоторых случаях они более эффективны, чем их длинные эквиваленты. По указанным причинам составные операции присваивания часто используются в профессионально написанных программах на Java.

Ниже приведен пример программы, в которой демонстрируются в действии несколько составных операций присваивания:

```
// Демонстрация ряда составных операций присваивания.
class OpEquals {
  public static void main(String[] args) {
    int a = 1;
    int b = 2;
    int c = 3;

    a += 5;
    b *= 4;
    c += a * b;
    c %= 6;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}
```

Вот вывод программы:

```
a = 6
b = 8
c = 3
```

Операции инкремента и декремента

Операции инкремента и декремента в Java обозначаются с помощью ++ и --. Они были представлены в главе 2, а здесь будут обсуждаться более подробно. Как вы увидите, они обладают рядом особых характеристик, которые делают их довольно интересными. Начнем с рассмотрения того, что именно делают операции инкремента и декремента.

Операция инкремента увеличивает на единицу значение своего операнда, а операция декремента уменьшает на единицу значение своего операнда. Например, следующий оператор:

```
x = x + 1;
```

можно переписать с целью применения операции инкремента:

```
x++;
```

Аналогично представленный ниже оператор:

```
x = x - 1;
```

эквивалентен такому:

```
x--;
```

Приведенные операции уникальны тем, что могут встречаться и в *постфиксной* форме, когда они следуют за операндом, как было только что показано, и в *префиксной* форме, когда они предшествуют операнду. В предшествующих примерах никаких отличий между префиксной и постфиксной формами не было. Однако когда операторы инкремента и/или декремента входят в состав более крупного выражения, то между этими двумя формами проявляется тонкая, но важная разница. В префиксной форме операнд инкрементируется или декрементируется перед получением значения для использования в выражении. В постфиксной форме предыдущее значение извлекается для применения в выражении, после чего модифицируется операнд. Например:

```
x = 42;
y = ++x;
```

В этом случае значение переменной y устанавливается в 43, как и следовало ожидать, поскольку инкремент происходит *до того*, как x присваивается y. Таким образом, строка кода `y = ++x;` будет эквивалентом следующих двух операторов:

```
x = x + 1;
y = x;
```

Тем не менее, при такой записи:

```
x = 42;
y = x++;
```

значение x получается до выполнения оператора приращения, а потому значение y равно 42. Разумеется, в обоих случаях x устанавливается в 43. Строка `y = x++;` является эквивалентом двух операторов:

```
y = x;
x = x + 1;
```

В показанной ниже программе демонстрируется работа операции инкремента.

```
// Демонстрация работы ++.
class IncDec {
  public static void main(String[] args) {
    int a = 1;
    int b = 2;
    int c;
    int d;
    c = ++b;
    d = a++;
    c++;
```

```
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
  }
}
```

В результате запуска программы вы получите следующий вывод:

```
a = 2
b = 3
c = 4
d = 1
```

Побитовые операции

В Java определено несколько *побитовых операций*, которые можно применять к целочисленным типам: `long`, `int`, `short`, `char` и `byte`. Такие операции воздействуют на отдельные биты своих операндов. Они перечислены в табл. 4.2.

Таблица 4.2. Побитовые операции языка Java

~	Побитовое унарное НЕ
&	Побитовое И
|	Побитовое ИЛИ
^	Побитовое исключающее ИЛИ
>>	Сдвиг вправо
>>>	Сдвиг вправо с заполнением нулями
<<	Сдвиг влево
&=	Побитовое И с присваиванием
|=	Побитовое ИЛИ с присваиванием
^=	Побитовое исключающее ИЛИ с присваиванием
>>=	Сдвиг вправо с присваиванием
>>>=	Сдвиг вправо с заполнением нулями и присваиванием
<<=	Сдвиг влево с присваиванием

Поскольку побитовые операции манипулируют битами внутри целого числа, важно понимать, какое влияние такие манипуляции могут оказать на значение. В частности, полезно знать, как в Java хранятся целочисленные значения и каким образом представляются отрицательные числа. Итак, прежде чем продолжить, давайте кратко обсудим эти две темы.

Все целочисленные типы представлены двоичными числами различной разрядности. Например, значение 42 типа `byte` в двоичном формате выгля-

дит как 00101010, где каждая позиция представляет собой степень двойки, начиная с 20 в крайнем правом бите. Следующей битовой позицией слева будет 21, или 2, далее влево будет 22, или 4, затем 8, 16, 32 и т.д. Таким образом, 42 имеет биты 1, установленные в позициях 1, 3 и 5 (считая от 0 справа); соответственно 42 — это сумма 21 + 23 + 25, что составляет 2 + 8 + 32.

Все целочисленные типы (кроме `char`) являются целыми числами со знаком, т.е. они могут представлять как положительные, так и отрицательные значения. В Java используется кодировка, известная как *дополнение до двух* или *дополнительный код*, которая предусматривает представление отрицательных чисел путем инвертирования (замены единиц на нули и наоборот) всех битов в значении и последующего добавления единицы к результату. Например, для представления –42 инвертируются все биты в 42, или 00101010, что дает 11010101, после чего к результату добавляется 1, давая в итоге 11010110, или –42. Чтобы декодировать отрицательное число, необходимо инвертировать все биты и добавить 1. Например, –42, или 11010110, в результате инвертирования дает 00101001, или 41, а после добавления 1 получается 42.

Причину, по которой в Java (и большинстве других языков программирования) применяется дополнение до двух, легко понять, если рассмотреть проблему *перехода через ноль*. Предполагая работу со значением типа `byte`, ноль представляется как 00000000. В дополнении до единицы простое инвертирование всех битов дает 11111111, что создает отрицательный ноль. Проблема в том, что в целочисленной математике отрицательный ноль недопустим. Эта проблема решается за счет использования дополнения до двух для представления отрицательных значений. Когда применяется дополнение до двух, к дополнению добавляется 1, что дает 100000000. Единичный бит оказывается слишком далеко слева и не умещается в значение `byte`, что приводит к желаемому поведению, где –0 совпадает с 0, а 11111111 является кодом для –1. Хотя в предыдущем примере мы использовали значение `byte`, тот же принцип применим ко всем целочисленным типам Java.

Из-за того, что в Java для хранения отрицательных чисел используется дополнение до двух и т.к. все целые числа представляют собой значения со знаком, применение побитовых операций может легко дать неожиданные результаты. Например, включение старшего бита приведет к тому, что результирующее значение будет интерпретироваться как отрицательное число, входило это в ваши намерения или нет. Во избежание неприятных сюрпризов просто помните, что старший бит определяет знак целого числа независимо от того, как он был установлен.

Побитовые логические операции

Существуют четыре побитовых логических операции: `&`, `|`, `^` и `~`. Результаты их выполнения приведены в табл. 4.3. В последующем обсуждении не забывайте о том, что побитовые операции применяются к каждому индивидуальному биту внутри каждого операнда.

Таблица 4.3. Результаты выполнения побитовых логических операций

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Побитовое НЕ

Унарная операция НЕ, ~, также называемая *побитовым дополнением*, инвертирует все биты своего операнда. Например, число 42, имеющее следующую битовую комбинацию:

```
00101010
```

после применения операции НЕ превращается в:

```
11010101
```

Побитовое И

Операция И, &, выдает единичный бит, если биты в обоих операндах также равны 1. В остальных случаях выдается нулевой бит. Вот пример:

```
  00101010  42
& 00001111  15
----------
  00001010  10
```

Побитовое ИЛИ

Операция ИЛИ, |, объединяет биты так, что если любой из битов в операндах равен 1, то результирующий бит будет единичным, как показано ниже:

```
  00101010  42
| 00001111  15
----------
  00101111  47
```

Побитовое исключающее ИЛИ

Операция исключающего ИЛИ (XOR), ^, объединяет биты таким образом, что если бит в одном операнде равен 1, то результирующий бит будет единичным. В противном случае результирующий бит будет нулевым. В представленном далее примере иллюстрируется действие операции ^. В этом примере также демонстрируется полезное характерное свойство операции XOR. Обратите внимание, как битовая комбинация 42 инвертируется везде, где второй операнд имеет единичный бит. Там, где второй операнд имеет нулевой бит, бит первого операнда не изменяется. Вы найдете такое свойство полезным при выполнении некоторых типов битовых манипуляций.

```
  00101010   42
^ 00001111   15
----------------
  00100101   37
```

Использование побитовых логических операций

Работа побитовых логических операций демонстрируется в следующей программе:

```
// Демонстрация работы побитовых логических операций.
class BitLogic {
  public static void main(String[] args) {
    String[] binary = {
      "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
      "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
    };
    int a = 3;  // 0 + 2 + 1 или 0011 в двоичной форме
    int b = 6;  // 4 + 2 + 0 или 0110 в двоичной форме
    int c = a | b;
    int d = a & b;
    int e = a ^ b;
    int f = (~a & b)|(a & ~b);
    int g = ~a & 0x0f;
    System.out.println("        a = " + binary[a]);
    System.out.println("        b = " + binary[b]);
    System.out.println("      a|b = " + binary[c]);
    System.out.println("      a&b = " + binary[d]);
    System.out.println("      a^b = " + binary[e]);
    System.out.println("~a&b|a&~b = " + binary[f]);
    System.out.println("       ~a = " + binary[g]);
  }
}
```

В этом примере a и b имеют битовые комбинации, которые представляют все четыре возможности для двух двоичных цифр: 0–0, 0–1, 1–0 и 1–1. Результаты в переменных c и d позволяют видеть работу с каждым битом операций | и &. Значения, присвоенные e и f, одинаковы и иллюстрируют работу операции ^. Массив строк по имени binary содержит удобочитаемое двоичное представление чисел от 0 до 15. Массив в примере индексируется так, чтобы показать двоичное представление каждого результата. Массив построен таким образом, что корректное строковое представление двоичного значения n хранится в binary[n]. Значение ~a объединяется посредством операции И с 0x0f (00001111 в двоичной форме) для его уменьшения до менее чем 16, чтобы его можно было вывести с использованием массива binary. Вот вывод программы:

```
        a = 0011
        b = 0110
      a|b = 0111
      a&b = 0010
      a^b = 0101
~a&b|a&~b = 0101
       ~a = 1100
```

Сдвиг влево

Операция сдвига влево, <<, сдвигает все биты значения влево на указанное количество позиций. Она имеет следующую общую форму:

значение << число

Здесь число устанавливает количество позиций для сдвига влево значения. То есть операция << перемещает все биты в указанном значении влево на количество битовых позиций, заданное в числе. При каждом сдвиге влево старший бит смещается (и утрачивается), а справа вставляется ноль. Это означает, что когда к операнду int применяется сдвиг влево, биты теряются, как только они сдвигаются за битовую позицию 31. В случае операнда типа long биты утрачиваются после битовой позиции 63.

При сдвиге значений byte и short автоматическое повышение типов в Java приводит к неожиданным результатам. Как вы знаете, во время вычисления выражения значения byte и short повышаются до int. Кроме того, результат такого выражения тоже имеет тип int. Это означает, что результатом сдвига влево значения byte или short будет значение int, а биты, сдвинутые влево, не будут утрачены до тех пор, пока они не сместятся за битовую позицию 31. Вдобавок отрицательное значение byte или short при повышении до int будет расширено знаком. Таким образом, старшие биты будут заполнены единицами. По указанным причинам выполнение сдвига влево значения byte или short подразумевает необходимость отбрасывания старших байтов из результата типа int. Например, при сдвиге влево значения byte оно сначала повышается до int, после чего сдвигается. Если нужно получить результат сдвинутого значения byte, то придется отбросить три старших байта результата. Самый простой способ решить задачу — преобразовать результат обратно в byte. Концепция демонстрируется в показанной далее программе:

```
// Сдвиг влево значения byte.
class ByteShift {
  public static void main(String[] args) {
    byte a = 64, b;
    int i;

    i = a << 2;
    b = (byte) (a << 2);

    System.out.println("Первоначальное значение a: " + a);
    System.out.println("i и b: " + i + " " + b);
  }
}
```

Программа генерирует следующий вывод:

```
Первоначальное значение a: 64
i и b: 256 0
```

Поскольку a повышается до int для целей вычисления, сдвиг влево значения 64 (0100 0000) дважды приводит к тому, что i содержит значение 256 (1 0000 0000). Однако значение в b содержит 0, т.к. после сдвига младший байт теперь равен нулю. Единственный единичный бит был сдвинут за его пределы.

Из-за того, что каждый сдвиг влево удваивает исходное значение, программисты часто используют данный факт как эффективную альтернативу умножению на 2. Но вам нужно быть начеку. Если вы сдвинете единичный бит в позицию старшего разряда (бит 31 или 63), то значение станет отрицательным, что иллюстрируется в следующей программе:

```
// Сдвиг влево как быстрый способ умножения на 2.
class MultByTwo {
  public static void main(String[] args) {
    int i;
    int num = 0xFFFFFFE;
    for(i=0; i<4; i++) {
      num = num << 1;
      System.out.println(num);
    }
  }
}
```

Вот вывод, который генерирует программа:

```
536870908
1073741816
2147483632
-32
```

Начальное значение было тщательно выбрано таким образом, чтобы после сдвига влево на 4 позиции бита получилось –32. Как видите, когда единичный бит сдвигается в позицию 31, число интерпретируется как отрицательное.

Сдвиг вправо

Операция сдвига вправо, >>, сдвигает все биты значения вправо на указанное количество позиций. Она имеет следующую общую форму:

значение >> *число*

Здесь число устанавливает количество позиций для сдвига вправо значения. То есть операция >> перемещает все биты в указанном значении вправо на количество битовых позиций, заданное в числе.

В показанном ниже фрагменте кода значение 32 сдвигается вправо на две позиции, приводя к тому, что a устанавливается в 8:

```
int a = 32;
a = a >> 2;  // a теперь содержит 8
```

Когда биты в значении смещаются за его пределы, то они утрачиваются. Например, в следующем фрагменте кода значение 35 сдвигается на две позиции вправо, что приводит к потере двух младших битов, в результате чего a снова получает значение 8:

```
int a = 35;
a = a >> 2;  // a содержит 8
```

Взглянув на ту же операцию в двоичной форме, становится ясно, как это происходит:

```
00100011   35
>> 2
00001000   8
```

Каждый раз, когда выполняется сдвиг значения вправо, оно делится на 2 с отбрасыванием любого остатка. В некоторых случаях данным фактом можно воспользоваться для высокопроизводительного целочисленного деления на 2.

При выполнении сдвига вправо старшие (крайние слева) биты, открытые сдвигом, заполняются предыдущим содержимым старшего бита. Это называется *расширением знака* и служит для сохранения знака отрицательных чисел при их сдвиге вправо. Например, –8 >> 1 равно –4, что в двоичном виде выглядит так:

```
11111000   -8
>> 1
11111100   -4
```

Интересно отметить, что в случае сдвига вправо значения –1 результат всегда остается –1, т.к. расширение знака будет приводить к большему количеству единиц в старших битах.

Иногда расширять знаки значений при сдвиге вправо нежелательно. Скажем, в приведенной далее программе значение `byte` преобразуется в шестнадцатеричное строковое представление. Обратите внимание, что сдвинутое значение маскируется с помощью операции И с `0x0f` для отбрасывания любых битов, дополненных знаком, чтобы значение можно было использовать в качестве индекса в массиве шестнадцатеричных символов.

```
// Маскирование расширения знака.
class HexByte {
  static public void main(String[] args) {
    char[] hex = {
      '0', '1', '2', '3', '4', '5', '6', '7',
      '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
    };

    byte b = (byte) 0xf1;

    System.out.println("b = 0x" + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
  }
}
```

Вот вывод программы:

```
b = 0xf1
```

Беззнаковый сдвиг вправо

Как только что объяснялось, операция >> автоматически заполняет старший бит его предыдущим содержимым каждый раз, когда происходит сдвиг, сохраняя знак значения. Тем не менее, иногда это нежелательно. Например, при выполнении сдвига чего-то, что не представляет собой числовое значение, расширение знака может оказаться ненужным. Такая ситуация часто

встречается при работе с пиксельными значениями и графикой. В ситуациях подобного рода обычно требуется помещать в старший бит ноль вне зависимости от того, каким было его начальное значение. Прием известен как *беззнаковый сдвиг* и предусматривает применение операции беззнакового сдвига вправо, >>>, которая всегда задвигает нули в старший бит.

В следующем фрагменте кода демонстрируется работа операции >>>. Здесь значение a устанавливается в -1, что приводит к установке всех 32 бит в 1. Затем значение сдвигается вправо на 24 позиции с заполнением старших 24 бит нулями и игнорированием расширения знака. В итоге a устанавливается в 255.

```
int a = -1;
a = a >>> 24;
```

Ниже показана та же самая операция в двоичной форме с целью дополнительной иллюстрации того, что происходит:

```
11111111 11111111 11111111 11111111  -1 в двоичном виде
>>>24
00000000 00000000 00000000 11111111  255 в двоичном виде
```

Операция >>> часто не настолько полезна, как хотелось бы, поскольку она имеет смысл только для 32- и 64-битных значений. Вспомните, что при вычислении выражений меньшие значения автоматически повышаются до int. В итоге происходит расширение знака, и сдвиг будет выполняться над 32-битным, а не 8- или 16-битным значением. То есть можно ожидать, что беззнаковый сдвиг вправо значения byte заполнит нулями, начиная с бита 7. Но это не так, поскольку на самом деле сдвигается 32-битное значение. Эффект демонстрируется в следующей программе:

```
// Беззнаковый сдвиг вправо значения типа byte.
class ByteUShift {
  static public void main(String[] args) {
    char[] hex = {
      '0', '1', '2', '3', '4', '5', '6', '7',
      '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
    };
    byte b = (byte) 0xf1;
    byte c = (byte) (b >> 4);
    byte d = (byte) (b >>> 4);
    byte e = (byte) ((b & 0xff) >> 4);

    System.out.println("           b = 0x"
      + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
    System.out.println("       b >> 4 = 0x"
      + hex[(c >> 4) & 0x0f] + hex[c & 0x0f]);
    System.out.println("      b >>> 4 = 0x"
      + hex[(d >> 4) & 0x0f] + hex[d & 0x0f]);
    System.out.println("(b & 0xff) >> 4 = 0x"
      + hex[(e >> 4) & 0x0f] + hex[e & 0x0f]);
  }
}
```

В показанном ниже выводе программы видно, что при работе с байтами операция >>> ничего не делает. Для данной демонстрации переменной b присваивается произвольное отрицательное значение типа byte. Затем переменной c присваивается значение b типа byte, сдвинутое вправо на четыре позиции, которое равно 0xff из-за ожидаемого расширения знака. Далее переменной d присваивается значение b типа byte с беззнаковым сдвигом вправо на четыре позиции, которым вопреки ожидаемому 0x0f будет 0xff по причине расширения знака, происшедшего при повышении b до int перед сдвигом. Последнее выражение устанавливает переменную e в значение b типа byte, маскированное до 8 бит с помощью операции И, после чего сдвинутое вправо на четыре позиции, что дает ожидаемое значение 0x0f. Обратите внимание, что операция беззнакового сдвига вправо не использовался для переменной d, т.к. состояние знакового бита после операции И было известно.

```
              b = 0xf1
         b >> 4 = 0xff
        b >>> 4 = 0xff
(b & 0xff) >> 4 = 0x0f
```

Составные побитовые операции присваивания

Все бинарные побитовые операции имеют составную форму, аналогичную форме алгебраических операций, которая сочетает в себе присваивание с побитовой операцией. Например, следующие два оператора, сдвигающие значение вправо на четыре позиции, эквивалентны:

```
a = a >> 4;
a >>= 4;
```

Аналогично два приведенных ниже оператора, присваивающие переменной a результат побитового выражения a ИЛИ b, тоже эквивалентны:

```
a = a | b;
a |= b;
```

В показанной далее программе создается несколько целочисленных переменных, с которыми затем осуществляются манипуляции с применением составных побитовых операций присваивания:

```
class OpBitEquals {
  public static void main(String[] args) {
    int a = 1;
    int b = 2;
    int c = 3;
    a |= 4;
    b >>= 1;
    c <<= 1;
    a ^= c;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}
```

Вот вывод программы:

```
a = 3
b = 1
c = 6
```

Операции отношения

Операции отношения устанавливают соотношение одного операнда с другим. В частности, они определяют равенство и порядок. Операции отношения описаны в табл. 4.4.

Таблица 4.4. Операции отношения языка Java

Операция	Описание
==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

Результатом этих операций является значение `boolean`. Операции отношения чаще всего используются в выражениях, управляющих оператором `if` и различными операторами циклов.

В Java значения любых типов, включая целые числа, числа с плавающей точкой, символы и булевские значения, можно сравнивать с помощью проверки на предмет равенства == и неравенства !=. Обратите внимание, что в Java равенство обозначается двумя знаками "равно", а не одним. (Помните: один знак "равно" — это операция присваивания.) С помощью операций определения порядка можно сравнивать только числовые типы, т.е. для выяснения, какой больше или меньше другого, допускается сравнивать лишь целочисленные операнды, операнды с плавающей точкой и символьные операнды.

Как уже упоминалось, результатом операции отношения будет значение `boolean`. Например, следующий фрагмент кода совершенно корректен:

```
int a = 4;
int b = 1;
boolean c = a < b;
```

В данном случае в переменной c сохраняется результат выполнения a<b (равный `false`).

Если ранее вы работали с C/C++, то обратите внимание, что в программах на C/C++ весьма распространены следующие виды операторов:

```
int done;
// ...
```

```
if(!done)...  // Допустимо в C/C++,
if(done)...   // но не в Java.
```

В коде Java такие операторы должны быть записаны так, как показано ниже:

```
if(done == 0)...  // Стиль Java.
if(done != 0)...
```

Причина в том, что истинное и ложное значения в Java определены не так, как в C/C++. В языках C и C++ истинным является любое ненулевое значение, а ложным — ноль. В Java `true` и `false` представляют собой нечисловые значения, которые никак не связаны с нулем или ненулевым значением. Следовательно, для проверки на предмет равенства или неравенства нулю должна явно применяться одна или несколько операций отношения.

Булевские логические операции

Булевские логические операции, перечисленные в табл. 4.5, работают только с операндами типа `boolean`. Все бинарные логические операции объединяют два значения `boolean`, чтобы сформировать результирующее значение `boolean`.

Таблица 4.5. Булевские логические операции языка Java

Операция	Описание
`&`	Логическое И
`|`	Логическое ИЛИ
`^`	Логическое исключающее ИЛИ
`||`	Короткозамкнутое ИЛИ
`&&`	Короткозамкнутое И
`!`	Логическое унарное НЕ
`&=`	Логическое И с присваиванием
`|=`	Логическое ИЛИ с присваиванием
`^=`	Логическое исключающее ИЛИ с присваиванием
`==`	Равно
`!=`	Не равно
`?:`	Тернарная операция "если–то–иначе"

Булевские логические операции `&`, `|` и `^` действуют на значениях типа `boolean` точно так же, как они работают с битами целого числа. Логическая операция `!` инвертирует булевское значение: `!true == false` и `!false == true`. В табл. 4.6 показано действие каждой булевской логической операции.

Таблица 4.6. Результаты выполнения булевских логических операций

A	B	A | B	A & B	A ^ B	!A
false	false	false	false	false	true
true	false	true	false	true	false
false	true	true	false	true	true
true	true	true	true	false	false

Ниже приведена программа, которая почти совпадает с представленным ранее примером `BitLogic`, но работает с логическими значениями типа `boolean`, а не с двоичными битами:

```
// Демонстрация работы булевских логических операций.
class BoolLogic {
  public static void main(String[] args) {
    boolean a = true;
    boolean b = false;
    boolean c = a | b;
    boolean d = a & b;
    boolean e = a ^ b;
    boolean f = (!a & b) | (a & !b);
    boolean g = !a;
    System.out.println("        a = " + a);
    System.out.println("        b = " + b);
    System.out.println("      a|b = " + c);
    System.out.println("      a&b = " + d);
    System.out.println("      a^b = " + e);
    System.out.println("!a&b|a&!b = " + f);
    System.out.println("       !a = " + g);
  }
}
```

После запуска программы вы увидите, что к значениям `boolean` применяются те же самые логические правила, что и к битам. Как видно в следующем выводе, строковое представление значения `boolean` является одним из литеральных значений `true` или `false`:

```
        a = true
        b = false
      a|b = true
      a&b = false
      a^b = true
!a&b|a&!b = true
       !a = false
```

Короткозамкнутые логические операции

В языке Java поддерживаются две интересные булевские операции, отсутствующие в ряде других языков программирования. Речь идет о вспомогательных версиях булевских операций И и ИЛИ, широко известные как *короткозамкнутые* логические операции. В табл. 4.6 легко заметить, что ре-

зультатом операции ИЛИ будет `true`, когда `A` имеет значение `true` вне зависимости от значения `B`. Аналогично операция И дает результат `false`, когда `A` равно `false` безотносительно к тому, какое значение имеет `B`. В случае применения форм `||` и `&&` вместо форм `|` и `&` указанных операций Java правый операнд не будет вычисляться в ситуации, когда результат выражения может быть определен только левым операндом. Это очень полезно, когда правильное функционирование предусматривает зависимость правого операнда от левого. В приведенном ниже фрагменте кода показано, как можно использовать в своих интересах короткозамкнутое логическое вычисление, чтобы убедиться в допустимости операции деления до ее выполнения:

```
if (denom != 0 && num / denom > 10)
```

Поскольку применяется короткозамкнутая форма операции И (`&&`), нет риска вызвать исключение во время выполнения, когда переменная `denom` равна нулю. Если бы данная строка кода была написана с использованием односимвольной версии операции `&`, то вычислялись бы обе стороны выражения, приводя к исключению во время выполнения, когда значение `denom` равно нулю.

Стандартная практика предусматривает применение короткозамкнутых форм И и ИЛИ в случаях, связанных с булевской логикой, оставляя односимвольные версии исключительно для побитовых операций. Однако из этого правила есть исключения. Например, взгляните на оператор следующего вида:

```
if(c==1 & e++ < 100) d = 100;
```

Здесь использование версии `&` гарантирует, что операция инкремента будет применена к `e` независимо от того, имеет переменная `c` значение `1` или нет.

На заметку! В формальной спецификации Java короткозамкнутые операции называются условным И и условным ИЛИ.

Операция присваивания

Операция присваивания использовалась, начиная с главы 2. Теперь пришло время взглянуть на нее формально. *Операция присваивания* обозначается одиночным знаком равенства (=) и работает в Java почти так же, как в любом другом языке программирования. Она имеет следующий общий вид:

```
переменная = выражение;
```

Тип переменной должен быть совместимым с типом выражения.

Операция присваивания обладает одной интересной особенностью, с которой вы, возможно, не знакомы: она позволяет создавать цепочку присваиваний. Например, взгляните на показанный ниже фрагмент кода:

```
int x, y, z;
x = y = z = 100;  // установить x, y и z в 100
```

Здесь переменные x, y и z устанавливаются в 100 с помощью единственного оператора. Прием работает из-за того, что операция = возвращает значение правого выражения. Таким образом, z = 100 дает значение 100, которое затем присваивается переменной y, а результат в свою очередь присваивается x. Применение "цепочки присваивания" — простой способ присвоить группе переменных общее значение.

Операция ?

В Java предусмотрена специальная *тернарная* операция, которая способна заменить определенные виды операторов "если–то–иначе". Она обозначается с помощью знака ?. Поначалу операция ? может показаться слегка сбивающей с толку, но после освоения ее можно использовать с высокой эффективностью. Вот общий вид операции ?:

выражение1 ? *выражение2* : *выражение3*

Здесь *выражение1* может быть любым выражением, результатом которого является значение boolean. Если результатом *выражения1* является true, тогда вычисляется *выражение2*, а иначе *выражение3*. Результатом операции ? будет результат вычисленного выражения. Типы результатов *выражения2* и *выражение3* должны быть одинаковыми (или совместимыми) и не могут быть void. Ниже показан пример применения операции ?:

```
ratio = denom == 0 ? 0 : num / denom;
```

При вычислении этого выражения присваивания сначала просматривается выражение *слева* от вопросительного знака. Если значение denom равно нулю, тогда вычисляется выражение *между* вопросительным знаком и двоеточием и используется в качестве значения всего выражения ?. Если значение denom не равно нулю, то вычисляется выражение *после* двоеточия и становится значением всего выражения ?. Затем результат операции ? присваивается переменной ratio.

В приведенной далее программе демонстрируется работа операции ?, которая применяется для получения абсолютной величины переменной:

```
// Демонстрация работы операции ?.
class Ternary {
  public static void main(String[] args) {
    int i, k;

    i = 10;
    k = i < 0 ? -i : i;  // получить абсолютную величину i
    System.out.print("Абсолютная величина ");
    System.out.println(i + " равна " + k);

    i = -10;
    k = i < 0 ? -i : i;  // получить абсолютную величину i
    System.out.print("Абсолютная величина ");
    System.out.println(i + " равна " + k);
  }
}
```

Вот вывод, сгенерированный программой:

```
Абсолютная величина 10 равна 10
Абсолютная величина -10 равна 10
```

Старшинство операций

В табл. 4.7 показан порядок старшинства операций Java от самого высокого до самого низкого. Операции в одной и той же строке имеют одинаковый приоритет. В бинарных операциях принят порядок вычисления слева направо (за исключением присваивания, которое вычисляется справа налево). Хотя формально `[]`, `()` и `.` являются разделителями, они также могут действовать как операции и в таком качестве обладать наивысшим приоритетом. Кроме того, обратите внимание на операцию стрелки (`->`), которая используется в лямбда-выражениях.

Таблица 4.7. Старшинство операций в Java

Высший приоритет						
++ (постфиксная)	-- (постфиксная)					
++ (префиксная)	-- (префиксная)	~	!	+ (унарная)	- (унарная)	(*приведение-типа*)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
&						
==	!=					
^						
|						
&&						
||						
?:						
->						
=	*операция*=					
Низший приоритет						

Использование круглых скобок

Круглые скобки повышают приоритет операций внутри них, что часто требуется для получения желаемого результата. Например, рассмотрим следующее выражение:

```
a >> b + 3
```

В показанном выражении сначала к значению b добавляется 3, после чего производится сдвиг a вправо на этот результат. То есть выражение можно переписать с применением избыточных круглых скобок:

```
a >> (b + 3)
```

Тем не менее, если сначала необходимо сдвинуть a вправо на b позиций и затем добавить к результату значение 3, тогда часть выражения понадобится заключить в скобки:

```
(a >> b) + 3
```

В дополнение к изменению обычного приоритета оператора, круглые скобки временами могут использоваться с целью прояснения смысла выражения. Для любого, кто читает ваш код, сложное выражение может оказаться трудным в понимании. Добавление избыточных, но уточняющих скобок к сложным выражениям может помочь предотвратить путаницу в дальнейшем. Например, какое из следующих выражений легче читать?

```
a | 4 + c >> b & 7
(a | (((4 + c) >> b) & 7))
```

И еще один момент: круглые скобки (избыточные или нет) не ухудшают производительность программы. Таким образом, добавление круглых скобок для уменьшения неоднозначности не повлияет отрицательно на вашу программу.

ГЛАВА

5 Управляющие операторы

Управляющие операторы используются в языке программирования для того, чтобы заставить поток выполнения продвигаться вперед и переходить в зависимости от изменений в состоянии программы. Управляющие операторы программы на Java можно разделить на следующие категории: выбор, итерация и переход. Операторы ***выбора*** позволяют программе выбирать различные пути выполнения на основе результата выражения или состояния переменной. Операторы ***итерации*** позволяют потоку выполнения повторять один или несколько операторов (т.е. операторы итерации образуют циклы). Операторы ***перехода*** позволяют программе выполняться нелинейным образом. Здесь рассматриваются все управляющие операторы Java.

Операторы выбора Java

В Java поддерживаются два оператора выбора: `if` и `switch`. Они предоставляют возможность управления потоком выполнения программы на основе условий, известных только во время выполнения. Вы будете приятно удивлены мощью и гибкостью этих двух операторов.

Оператор `if`

Оператор `if` был представлен в главе 2, а ниже он рассматривается более подробно. Оператор `if` является оператором условного перехода Java. Его можно применять для направления потока выполнения программы по двум разным путям. Вот его общая форма:

```
if(условие) оператор1;
else оператор2;
```

Здесь как *оператор1*, так и *оператор2* может быть одиночным оператором или составным оператором, заключенным в фигурные скобки (т.е. ***блоком***). В качестве *условия* может использоваться любое выражение, которое возвращает значение типа `boolean`. Конструкция `else` необязательна.

Оператор `if` работает следующим образом: если *условие* истинно, тогда выполняется *оператор1*. В противном случае выполняется *оператор2*

(если он присутствует). Ни в коем случае не будут выполнены оба оператора. Например, взгляните на приведенный далее код:

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

Если значение a меньше значения b, то a устанавливается в ноль, а в противном случае b устанавливается в ноль. Но никогда обе переменные не будут установлены в ноль.

Чаще всего выражение, применяемое для управления `if`, будет включать операторы отношения. Однако это не является формально необходимым. Управлять оператором `if` можно с помощью одной переменной типа `boolean`, как показано в следующем фрагменте кода:

```
boolean dataAvailable;
// ...
if(dataAvailable)
  ProcessData();
else
  waitForMoreData();
```

Помните, что непосредственно после `if` или `else` может находиться только один оператор. Если вы хотите включить больше операторов, тогда понадобится создать блок, как в приведенном далее фрагменте кода:

```
int bytesAvailable;
// ...
if(bytesAvailable > 0) {
  ProcessData();
  bytesAvailable -= n;
} else
  waitForMoreData();
```

Оба внутри блока `if` будут выполняться, если значение переменной `bytesAvailable` больше нуля.

Некоторым программистам удобно использовать фигурные скобки в операторе `if`, даже если в каждой конструкции имеется только один оператор. Такой прием позволяет легко добавлять еще один оператор позже, и не придется беспокоиться о том, что вы забудете о фигурных скобках. На самом деле частая причина ошибок связана с тем, что забывают определить блок, когда он нужен. Например, рассмотрим следующий фрагмент кода:

```
int bytesAvailable;
// ...
if(bytesAvailable > 0) {
  ProcessData();
  bytesAvailable -= n;
} else
  waitForMoreData();
  bytesAvailable = n;
```

Кажется вполне очевидным, что из-за уровня отступа оператор `bytesAvailable = n;` предназначался для выполнения внутри конструкции `else`. Тем не менее, как вы помните, пробелы в Java несущественны, и компилятор не может узнать, что именно было задумано. Код скомпилируется без ошибок, но при запуске он будет вести себя некорректно. Вот как исправить предыдущий пример:

```
int bytesAvailable;
// ...
if(bytesAvailable > 0) {
  ProcessData();
  bytesAvailable -= n;
} else {
  waitForMoreData();
  bytesAvailable = n;
}
```

Вложенные операторы `if`

Вложенный оператор `if` представляет собой оператор `if`, который является целью другого оператора `if` или `else`. Вложенные операторы `if` очень распространены в программировании. При вложении `if` главное помнить, что оператор `else` всегда ссылается на ближайший оператор `if`, который находится в том же блоке, что и `else`, и еще не связан с оператором `else`, например:

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d; // этот оператор if ассоциирован
  else a = c;        // с этим else
}
else a = d;          // этот оператор else относится к if(i == 10)
```

Как видно в комментариях, финальный оператор `else` не связан с `if(j < 20)`, поскольку он не находится в том же блоке (несмотря на то, что он — ближайший `if` без `else`). Взамен финальный оператор `else` связан с `if(i == 10)`. Внутренний оператор `else` относится к `if(k > 100)`, потому что он — ближайший `if` в том же блоке.

Цепочка `if-else-if`

Распространенной программной конструкцией, основанной на последовательности вложенных операторов `if`, является цепочка `if-else-if`, которая выглядит следующим образом:

```
if(условие)
  оператор;
else if(условие)
  оператор;
else if(условие)
  оператор;
.
.
.
else
  оператор;
```

Операторы if выполняются сверху вниз. Как только одно из условий, управляющих if, становится истинным, выполняется оператор, связанный с этим if, и остальная часть цепочки игнорируется. Если ни одно из условий не выполняется, тогда будет выполнен последний оператор else. Финальный оператор else действует как условие по умолчанию; т.е. если все другие условные проверки не пройдены, то выполняется последний оператор else. Если финального оператора else нет и все остальные условия ложны, тогда никакие действия не предпринимаются.

Ниже приведена программа, в которой цепочка if-else-if применяется для определения, к какому времени года относится специфический месяц:

```
// Демонстрация операторов if-else-if.
class IfElse {
  public static void main(String[] args) {
    int month = 4;  // апрель
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "зима";
    else if(month == 3 || month == 4 || month == 5)
      season = "весна";
    else if(month == 6 || month == 7 || month == 8)
      season = "лето";
    else if(month == 9 || month == 10 || month == 11)
      season = "осень";
    else
      season = "Несуществующий месяц";

    System.out.println("Апрель - это " + season + ".");
                    // Принадлежность апреля ко времени года
  }
}
```

Вот вывод, генерируемый программой:

```
Апрель - это весна.
```

Возможно, вы захотите поэкспериментировать с этой программой, прежде чем двигаться дальше. Как вы обнаружите, независимо от того, какое значение вы присвоите месяцу, будет выполнен один и только один оператор присваивания в цепочке.

Традиционный оператор switch

Оператор switch в Java обеспечивает переход по множеству путей. Он предлагает простой способ направления потока выполнения на разные части кода в зависимости от значения выражения. Таким образом, он часто оказывается лучшей альтернативой большому набору операторов if-else-if.

Первым делом необходимо отметить, что начиная с JDK 14, оператор switch был значительно улучшен и расширен рядом новых возможностей, выходящих далеко за рамки его традиционной формы. Традиционная форма switch была частью Java с самого начала и потому используется весьма

широко. Кроме того, эта форма будет работать во всех средах разработки Java для всех читателей. Из-за существенного характера последних улучшений switch они описаны в главе 17 в контексте других недавних дополнений языка Java. Здесь рассматривается традиционная форма оператора switch, которая имеет следующий вид:

```
switch(выражение) {
  case значение1:
    // последовательность операторов
    break;
  case значение2:
    // последовательность операторов
    break;
  .
  .
  .
  case значениеN:
    // последовательность операторов
    break;
  default:
    // стандартная последовательность операторов
}
```

В версиях Java, предшествующих JDK 7, выражение должно давать значение типа byte, short, int, char или перечисления. (Перечисления описаны в главе 12.) В настоящее время *выражение* также может иметь тип String. Каждое значение, указанное в операторах case, должно быть уникальным константным выражением (например, литеральным значением). Дублирование значений в операторах case не разрешено. Тип каждого значения должен быть совместимым с типом *выражения*.

Вот как функционирует традиционный оператор switch: значение *выражения* сравнивается с каждым значением в операторах case. Если совпадение найдено, то выполняется кодовая последовательность, следующая за оператором case. Если ни одна из констант не соответствует значению *выражения*, тогда выполняется оператор default. Однако оператор default необязателен. Если ни один из операторов case не дает совпадения, а оператор default отсутствует, то дальнейшие действия не предпринимаются.

Оператор break применяется внутри switch для завершения последовательности операторов. Когда встречается оператор break, поток выполнения переходит к первой строке кода, следующей за полным оператором switch, обеспечивая эффект "выпрыгивания" из switch.

Ниже показан простой пример использования оператора switch:

```
// Простой пример применения оператора switch.
class SampleSwitch {
  public static void main(String[] args) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i равно нулю.");
          break;
```

```
        case 1:
          System.out.println("i равно одному.");
          break;
        case 2:
          System.out.println("i равно двум.");
          break;
        case 3:
          System.out.println("i равно трем.");
          break;
        default:
          System.out.println("i больше трех.");
      }
    }
}
```

Программа генерирует следующий вывод:

```
i равно нулю.
i равно одному.
i равно двум.
i равно трем.
i больше трех.
i больше трех.
```

Как видите, каждый раз в цикле выполняются операторы, связанные с константой в `case`, которая совпадает со значением `i`, а все остальные пропускаются. После того, как значение `i` становится больше 3, операторы `case` не дают совпадения, поэтому выполняется оператор `default`. Оператор `break` необязателен. Если опустить `break`, то выполнение продолжится в следующем `case`. Иногда желательно иметь несколько `case` без операторов `break` между ними. Например, рассмотрим приведенную далее программу:

```
// В операторе switch операторы break необязательны.
class MissingBreak {
  public static void main(String[] args) {
    for(int i=0; i<12; i++)
      switch(i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i меньше 5.");
          break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println("i меньше 10.");
          break;
        default:
          System.out.println("i больше или равно 10.");
      }
    }
}
```

Программа генерирует такой вывод:

```
i меньше 5.
i меньше 5.
i меньше 5.
i меньше 5.
i меньше 5.
i меньше 10.
i меньше 10.
i меньше 10.
i меньше 10.
i меньше 10.
i больше или равно 10.
i больше или равно 10.
```

Легко заметить, что поток выполнения проходит через все операторы case, пока не доберется до оператора case (или до конца switch).

Хотя предыдущий пример, конечно же, придуман в целях иллюстрации, пропуск оператора break имеет много практических применений в реальных программах. Ниже демонстрируется его более реалистичное использование в переработанной программе определения принадлежности месяца к временам года. Теперь в нем применяется оператор switch для обеспечения более эффективной реализации.

```
// Усовершенствованная версия программы, определяющей принадлежность
// месяца к времени года.
class Switch {
  public static void main(String[] args) {
    int month = 4;  // апрель
    String season;
    switch (month) {
      case 12:
      case 1:
      case 2:
        season = "зима";
        break;
      case 3:
      case 4:
      case 5:
        season = "весна";
        break;
      case 6:
      case 7:
      case 8:
        season = "лето";
        break;
      case 9:
      case 10:
      case 11:
        season = "осень";
        break;
      default:
        season = "несуществующий месяц";
```

```
    System.out.println("Апрель - это " + season + ".");
  }
}
```

Как уже упоминалось, для управления оператором `switch` можно также использовать строку:

```
// Применение строки для управления оператором switch.
class StringSwitch {
  public static void main(String[] args) {
    String str = "two";

    switch(str) {
      case "one":
        System.out.println("один");
        break;
      case "two":
        System.out.println("два");
        break;
      case "three":
        System.out.println("три");
        break;
      default:
        System.out.println("совпадений нет");
        break;
    }
  }
}
```

Вывод, сгенерированный программой, вполне ожидаем:

```
два
```

Строка, содержащаяся в `str` (`"two"` в рассматриваемой программе), проверяется на предмет соответствия константам в `case`. Когда совпадение найдено (во втором операторе `case`), выполняется кодовая последовательность, связанная с данным `case`.

Возможность применения строк в операторе `switch` упрощает код во многих ситуациях. Скажем, использовать `switch` на основе строк эффективнее по сравнению с эквивалентной последовательностью операторов `if/else`. Тем не менее, выполнение оператора `switch` по строкам может оказаться более затратным, чем по целым числам. Таким образом, лучше всего применять `switch` на основе строк только в тех случаях, когда управляющие данные уже представлены в строковой форме. Другими словами, не используйте строки в `switch` без настоятельной необходимости.

Вложенные операторы `switch`

Вы можете применять `switch` как часть последовательности операторов внешнего `switch`. Он называется вложенным оператором `switch`. Поскольку оператор `switch` определяет собственный блок, между константами `case` во внутреннем `switch` и во внешнем `switch` конфликты не возникают. Например, следующий фрагмент кода совершенно допустим:

```
switch(count) {
  case 1:
    switch(target) {  // вложенный switch
      case 0:
        System.out.println("target равно нулю");
        break;
      case 1:        // никаких конфликтов с внешним switch
        System.out.println("target равно одному");
        break;
    }
    break;
  case 2:           // ...
```

Здесь оператор `case 1:` во внутреннем `switch` не конфликтует с оператором `case 1:` во внешнем `switch`. Переменная `count` сравнивается только со списком значений в операторах `case` на внешнем уровне. Если значение `count` равно 1, то `target` сравнивается со значениями в операторах `case` внутреннего списка.

Подводя итоги, важно отметить три особенности оператора `switch`.

- Оператор `switch` отличается от `if` тем, что он может проверять только на предмет равенства, тогда как оператор `if` способен оценивать логическое выражение любого вида. То есть `switch` ищет только совпадение значения выражения с одной из констант в операторах `case`.
- Никакие две константы `case` в одном `switch` не могут иметь одинаковые значения. Разумеется, один оператор `switch` и включающий его внешний `switch` могут иметь общие константы `case`.
- Оператор `switch` обычно более эффективен, чем набор вложенных операторов `if`.

Последний пункт особенно интересен, потому что он дает представление о том, как работает компилятор Java. При компиляции оператора `switch` компилятор Java проверит каждую константу `case` и создаст "таблицу переходов", которую будет использовать для выбора пути выполнения в зависимости от значения выражения. Следовательно, если вам нужно делать выбор среди большой группы значений, то оператор `switch` будет работать намного быстрее, чем эквивалентная логика, реализованная с применением последовательности `if-else`. Компилятор способен добиться этого, т.к. ему известно, что все константы `case` имеют один и тот же тип и просто должны сравниваться на равенство с выражением `switch`. Что касается подобного знания длинного списка выражений `if`, то компилятор им не располагает.

Помните! Недавно возможности и характеристики оператора `switch` были существенно расширены по сравнению с только что описанной традиционной формой `switch`. Усовершенствованный оператор `switch` рассматривается в главе 17.

Операторы итерации

В Java доступны операторы итерации `for`, `while` и `do-while`. Они создают то, что мы обычно называем *циклами*. Как вы наверняка знаете, цикл многократно выполняет один и тот же набор инструкций, пока не будет удовлетворено условие завершения. Вы увидите, что язык Java предлагает циклы, подходящие для любых нужд программирования.

Цикл `while`

Цикл `while` — самый фундаментальный оператор итерации в Java. Он повторяет выполнение оператора или блока до тех пор, пока истинно управляющее выражение. Вот его общая форма:

```
while(условие) {
  // тело цикла
}
```

Здесь *условием* может быть любое булевское выражение. Тело цикла будет выполняться до тех пор, пока условное выражение истинно. Когда *условие* становится ложным, управление переходит на строку кода, непосредственно следующую за циклом. Фигурные скобки не обязательны, если повторяться должен только один оператор.

Ниже показан цикл `while`, который ведет обратный отсчет от 10, выводя ровно десять строк "импульсов":

```
// Демонстрация работы цикла while.
class While {
  public static void main(String[] args) {
    int n = 10;

    while(n > 0) {
      System.out.println("Импульс номер " + n);
      n--;
    }
  }
}
```

Вот результат запуска программы:

```
Импульс номер 10
Импульс номер 9
Импульс номер 8
Импульс номер 7
Импульс номер 6
Импульс номер 5
Импульс номер 4
Импульс номер 3
Импульс номер 2
Импульс номер 1
```

Поскольку условное выражение цикла `while` вычисляется в начале цикла, тело цикла не выполнится ни разу, если условие ложно изначально. Например, в следующем фрагменте кода вызов `println()` никогда не выполняется:

```
int a = 10, b = 20;
while(a > b)
  System.out.println("Это никогда не отобразится");
```

Тело цикла while (или любого другого цикла Java) может быть пустым. Дело в том, что в Java синтаксически допустим пустой оператор (состоящий только из точки с запятой). Взгляните на следующую программу:

```
// Тело цикла может быть пустым.
class NoBody {
  public static void main(String[] args) {
    int i, j;

    i = 100;
    j = 200;

    // найти среднюю точку между i и j
    while(++i < --j);  // тело в цикле отсутствует

    System.out.println("Средняя точка равна " + i);
  }
}
```

Программа находит среднюю точку между i и j. Она генерирует следующий вывод:

```
Средняя точка равна 150
```

Рассмотрим, как работает этот цикл while. Значение i инкрементируется, а значение j декрементируется. Затем значения i и j сравниваются друг с другом. Если новое значение i все еще меньше нового значения j, тогда цикл повторяется. Если значение i больше или равно значения j, то цикл останавливается. После выхода из цикла переменная i будет хранить значение, которое находится посередине между исходными значениями i и j. (Конечно, такая процедура работает только тогда, когда i меньше j.) Понятно, что здесь нет необходимости иметь тело цикла; все действие происходит внутри самого условного выражения. В профессионально написанном коде на Java короткие циклы часто кодируются без тел, когда управляющее выражение способно самостоятельно обрабатывать все детали.

Цикл do-while

Как вы только что видели, если условное выражение, управляющее циклом while, изначально ложно, то тело цикла вообще не будет выполнено. Однако иногда тело цикла желательно выполнить хотя бы один раз, даже если изначально условное выражение ложно. Другими словами, бывают случаи, когда вы хотели бы проверять выражение завершения в конце цикла, а не в его начале. К счастью, в Java есть цикл, который делает именно это: do-while. Цикл do-while всегда выполняет свое тело, по крайней мере, один раз, поскольку его условное выражение находится в конце цикла. Вот его общая форма:

```
do {
  // тело цикла
} while(условие);
```

На каждой итерации цикла do-while сначала выполняется тело цикла, а затем вычисляется условное выражение. При истинном значении выражения цикл будет повторяться. В противном случае цикл завершается. Как и во всех циклах Java, *условие* должно быть булевским выражением.

Ниже показана переработанная версия программы, организующей выдачу "импульсов", в которой демонстрируется использование цикла do-while. Она выдает тот же вывод, что и ранее.

```
// Демонстрация работы цикла do-while.
class DoWhile {
  public static void main(String[] args) {
    int n = 10;

    do {
      System.out.println("Импульс номер " + n);
      n--;
    } while(n > 0);
  }
}
```

Цикл в предыдущей программе, хотя и будучи формально корректным, может быть записан более эффективно следующим образом:

```
do {
  System.out.println("Импульс номер " + n);
} while(--n > 0);
```

В приведенном выше примере выражение (--n > 0) объединяет декремент n и проверку на равенство нулю в одно выражение. Вот как оно работает. Сначала выполняется операция --n, декрементирующая n и возвращающая новое значение n, которое затем сравнивается с нулем. Если оно больше нуля, тогда цикл продолжается; в противном случае цикл завершается.

Цикл do-while особенно полезен при обработке выбора пункта меню, потому что обычно требуется, чтобы тело цикла меню выполнялось хотя бы один раз. В представленной далее программе реализована очень простая справочная система для операторов выбора и итерации языка Java:

```
// Использование цикла do-while для обработки выбора пункта меню.
class Menu {
  public static void main(String[] args)
    throws java.io.IOException {
    char choice;

    do {
      System.out.println("Краткая справка по: ");
      System.out.println("  1. if");
      System.out.println("  2. switch");
      System.out.println("  3. while");
      System.out.println("  4. do-while");
      System.out.println("  5. for\n");
      System.out.println("Выберите вариант:");
      choice = (char) System.in.read();
    } while( choice < '1' || choice > '5');

    System.out.println("\n");
```

```
    switch(choice) {
      case '1':
        System.out.println("Оператор if:\n");
        System.out.println("if(условие) оператор;");
        System.out.println("else оператор;");
        break;
      case '2':
        System.out.println("Оператор switch:\n");
        System.out.println("switch(выражение) {");
        System.out.println("  case константа:");
        System.out.println("    последовательность операторов");
        System.out.println("    break;");
        System.out.println("  //...");
        System.out.println("}");
        break;
      case '3':
        System.out.println("Оператор while:\n");
        System.out.println("while(условие) оператор;");
        break;
      case '4':
        System.out.println("Оператор do-while:\n");
        System.out.println("do {");
        System.out.println("  оператор;");
        System.out.println("} while(условие);");
        break;
      case '5':
        System.out.println("Оператор for:\n");
        System.out.print("for(инициализация; условие; итерация)");
        System.out.println("  оператор;");
        break;
    }
  }
}
```

Вот пример вывода, генерируемого программой:

```
Краткая справка по:
  1. if
  2. switch
  3. while
  4. do-while
  5. for
Выберите вариант:
4
Оператор do-while:
do {
  оператор;
} while(условие);
```

Цикл `do-while` применяется в программе для проверки того, что пользователь ввел правильный вариант. Если введенный вариант некорректен, тогда пользователю выдается повторный запрос. Поскольку меню должно отображаться хотя бы один раз, цикл `do-while` идеально подходит для решения такой задачи.

Есть еще несколько замечаний по поводу рассмотренного примера. Обратите внимание, что символы читаются с клавиатуры с помощью вызова `System.in.read()`. Это одна из функций консольного ввода Java. Хотя консольные методы ввода-вывода Java будут подробно обсуждаться лишь в главе 13, сейчас следует отметить, что для получения выбранного пользователем варианта используется вызов метода `System.in.read()`, который читает символы из стандартного ввода (возвращаемые в виде целых чисел, по причине чего возвращаемое значение было преобразовано в `char`). По умолчанию стандартный ввод буферизируется построчно, так что пользователю придется нажать клавишу <Enter>, прежде чем любые введенные им символы отправятся в программу.

Консольный ввод Java может оказаться немного неудобным для работы. Кроме того, в большинстве реальных программ на Java будет реализован графический пользовательский интерфейс. По упомянутой причине в примерах, приводимых в книге, консольный ввод применяется нечасто. Тем не менее, в таком контексте он полезен. Еще один момент, который следует учитывать: поскольку используется метод `System.in.read()`, в программе должна быть указана конструкция `throws java.io.IOException`, которая необходима для обработки ошибок ввода. Это часть средств обработки исключений Java, которые обсуждаются в главе 10.

Цикл `for`

Простая форма цикла `for` была представлена в главе 2. Вы увидите, что цикл `for` является мощной и универсальной конструкцией.

Существуют две формы цикла `for`. Первая — это традиционная форма, которая применялась со времен первоначальной версии Java, а вторая — более новая форма в стиле "for-each", появившаяся в JDK 5. Мы обсудим оба типа циклов `for`, начав с традиционной формы.

Общий вид традиционной формы оператора `for` выглядит так:

```
for(инициализация; условие; итерация) {
  // тело цикла
}
```

Если многократно выполнять требуется лишь один оператор, то фигурные скобки не нужны.

Рассмотрим работу цикла `for`. При первом запуске цикла выполняется часть *инициализация* цикла. В общем случае она представляет собой выражение, которое устанавливает значение *переменной управления циклом*, которая действует в качестве счетчика, управляющего циклом. Важно понимать, что инициализирующее выражение выполняется только один раз. Затем вычисляется *условие*, которое должно быть булевским выражением. Как правило, оно сравнивает переменную управления циклом с целевым значением. Если *условие* истинно, тогда выполняется тело цикла, а если ложно, то цикл завершается. Далее выполняется часть *итерация* цикла, которая обычно является выражением, инкрементирующим или декрементирующим перемен-

ную управления циклом. После этого цикл повторяется, при каждом проходе вычисляя условное выражение, выполняя тело цикла и вычисляя выражение итерации. Такой процесс происходит до тех пор, пока управляющее выражение не станет ложным.

Вот версия программы, организующей выдачу "импульсов", в которой используется цикл `for`:

```
// Демонстрация работы цикла for.
class ForTick {
  public static void main(String[] args) {
    int n;

    for(n=10; n>0; n--)
      System.out.println("Импульс номер " + n);
  }
}
```

Объявление переменной управления циклом внутри цикла `for`

Часто переменная, управляющая циклом `for`, необходима только для целей цикла и больше нигде не задействована. В таком случае переменную можно объявлять внутри части *инициализация* цикла `for`. Например, ниже показано, как переписать код предыдущей программы, чтобы объявить переменную управления циклом `n` типа `int` внутри `for`:

```
// Объявление переменной управления циклом внутри for.
class ForTick {
  public static void main(String[] args) {

    // здесь переменная is объявляется внутри цикла for
    for(int n=10; n>0; n--)
      System.out.println("Импульс номер " + n);
  }
}
```

При объявлении переменной внутри цикла `for` следует помнить об одном важном аспекте: область видимости этой переменной ограничена циклом `for`. За пределами цикла `for` переменная перестает существовать. Если переменную управления циклом необходимо применять в другом месте программы, тогда объявлять ее внутри оператора `for` нельзя.

В ситуации, когда переменная управления циклом больше нигде не нужна, большинство программистов на Java объявляют ее внутри `for`. Например, далее приведена простая программа, которая проверяет, является ли число простым. Обратите внимание, что переменная управления циклом `i` объявлена внутри `for`, т.к. в других местах она не используется.

```
// Проверка, является ли число простым.
class FindPrime {
  public static void main(String[] args) {
    int num;
    boolean isPrime;

    num = 14;
```

```
    if(num < 2) isPrime = false;
    else isPrime = true;

    for(int i=2; i <= num/i; i++) {
      if((num % i) == 0) {
        isPrime = false;
        break;
      }
    }

    if(isPrime) System.out.println("является простым");
    else System.out.println("не является простым");
  }
}
```

Использование запятой

Временами желательно включить более одного оператора в части *инициализация* и *итерация* цикла `for`. Например, взгляните на цикл в следующей программе:

```
class Sample {
  public static void main(String[] args) {
    int a, b;

    b = 4;
    for(a=1; a<b; a++) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
      b--;
    }
  }
}
```

Как видите, цикл управляется взаимодействием двух переменных. Поскольку цикл управляется двумя переменными, было бы удобно поместить их внутрь оператора `for`, а не обрабатывать `b` вручную. К счастью, в Java есть способ решить проблему. Чтобы позволить двум и более переменным управлять циклом `for`, в части *инициализация* и *итерация* цикла `for` разрешено включать несколько операторов, разделяя их запятыми.

С применением запятой предыдущий цикл `for` можно реализовать более эффективно:

```
// Использование запятой.
class Comma {
  public static void main(String[] args) {
    int a, b;

    for(a=1, b=4; a<b; a++, b--) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
    }
  }
}
```

Здесь в части *инициализация* устанавливаются значения как a, так и b. Два разделенных запятыми оператора в части *итерация* выполняются на каждой итерации цикла. Программа генерирует следующий вывод:

```
a = 1
b = 4
a = 2
b = 3
```

Некоторые разновидности цикла `for`

Существует несколько разновидностей цикла `for`, расширяющих его возможности и применимость. Причина такой гибкости связана с тем, что три части цикла `for` — *инициализация*, *условие* и *итерация* — не обязательно использовать только по прямому назначению. В действительности три части `for` можно применять для любых желаемых целей. Давайте рассмотрим примеры.

Один из наиболее распространенных вариантов задействует условное выражение. В частности, этому выражению не требуется сравнивать переменную управления циклом с некоторым значением. На самом деле условием, управляющим `for`, может быть любое булевское выражение. Например, взгляните на показанный ниже фрагмент кода:

```
boolean done = false;

for(int i=1; !done; i++) {
  // ...
  if(interrupted()) done = true;
}
```

В приведенном примере цикл `for` продолжает выполняться до тех пор, пока переменная `done` типа `boolean` не будет установлена в `true`. Значение `i` в нем не проверяется.

Есть еще одна интересная разновидность цикла `for`, в которой может отсутствовать либо часть *инициализация*, либо часть *итерация*, либо то и другое, как демонстрируется в следующей программе:

```
// Части цикла for могут быть пустыми.
class ForVar {
  public static void main(String[] args) {
    int i;
    boolean done = false;

    i = 0;
    for( ; !done; ) {
      System.out.println("i равно " + i);
      if(i == 10) done = true;
      i++;
    }
  }
}
```

Здесь выражения инициализации и итерации вынесены за пределы цикла `for`. Таким образом, части `for` пусты. Хотя в настолько простом примере это не имеет значения (более того, подобный стиль будет считаться довольно плохим), могут возникать ситуации, когда такой подход имеет смысл. Например, если начальное условие задается с помощью сложного выражения в другом месте программы или переменная управления циклом изменяется в зависимости от действий, происходящих в теле цикла, то может быть уместно оставить данные части цикла `for` пустыми.

Рассмотрим еще одну разновидность цикла `for`. Можно преднамеренно создать бесконечный цикл (цикл, который никогда не завершится), если оставить все три части `for` пустыми. Вот пример:

```
for( ; ; ) {
  // ...
}
```

Этот цикл будет работать нескончаемо долго, потому что нет условия, при котором он завершится. Хотя некоторые программы, подобные командным процессорам операционной системы, требуют бесконечного цикла, большинство "бесконечных циклов" на самом деле представляют собой просто циклы со специальными требованиями относительно завершения. Вскоре вы увидите, что существует способ завершить цикл (даже бесконечный вроде показанного выше), в котором отсутствует нормальное условное выражение.

Версия цикла `for` в стиле "for-each"

Вторая форма `for` реализует цикл в стиле "for-each". Вероятно, вам известно, что в современной теории языков программирования была принята концепция "for-each", которая стала стандартным функциональным средством, ожидаемым программистами. Цикл в стиле "for-each" предназначен для прохода по коллекции объектов, такой как массив, строго последовательно от начала до конца. В Java стиль "for-each" также называют *расширенным* циклом `for`.

Общая форма версии "for-each" цикла `for` выглядит следующим образом:

```
for(тип переменная-итерации : коллекция) блок-операторов
```

Здесь `тип` указывает тип, а *переменная-итерации* — имя переменной итерации, которая будет получать элементы из коллекции по одному за раз, от начала до конца. Коллекция, по которой проходит цикл, указывается в *коллекции*. Существуют различные типы коллекций, которые можно использовать с `for`, но в настоящей главе применяется только массив. (Другие типы коллекций, которые можно использовать с `for`, вроде тех, что определены в Collections Framework, обсуждаются далее в книге.) На каждой итерации цикла из коллекции извлекается очередной элемент и сохраняется в *переменной-итерации*. Цикл повторяется до тех пор, пока не будут получены все элементы коллекции.

Поскольку переменная итерации получает значения из коллекции, *тип* обязан совпадать или быть совместимым с типом элементов, хранящихся в

коллекции. Таким образом, при проходе по массивам тип должен быть совместимым с типом элементов массива.

Чтобы понять мотивы создания цикла в стиле "for-each", рассмотрим тип цикла `for`, который он призван заменить. В следующем фрагменте кода для вычисления суммы значений в массиве применяется традиционный цикл `for`:

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int i=0; i < 10; i++) sum += nums[i];
```

Для вычисления суммы все элементы в `nums` читаются по порядку от начала до конца, т.е. целый массив читается в строго последовательном порядке. Это достигается путем ручной индексации массива `nums` с использованием переменной управления циклом по имени `i`.

Стиль "for-each" цикла `for` позволяет автоматизировать предыдущий цикл. В частности, он избавляет от необходимости устанавливать счетчик циклов, указывать начальное и конечное значение и вручную индексировать массив. Взамен он автоматически проходит по всему массиву, получая по одному элементу за раз, последовательно, от начала до конца. Например, вот предыдущий фрагмент, переписанный с применением версии цикла `for` в стиле "for-each":

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int x: nums) sum += x;
```

При каждом проходе цикла переменной `x` автоматически присваивается значение, равное очередному элементу в `nums`. Таким образом, на первой итерации `x` содержит 1, на второй — 2 и т.д. Синтаксис не только стал проще, но он также предотвращает возникновение ошибок выхода за границы массива.

Ниже показана полная программа, в которой демонстрируется только что описанная версия цикла `for` в стиле "for-each":

```
// Использование цикла for в стиле "for-each".
class ForEach {
  public static void main(String[] args) {
    int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int sum = 0;

    // Применить цикл for в стиле "for-each" для отображения
    // и суммирования значений.
    for(int x : nums) {
      System.out.println("Значение: " + x);
      sum += x;
    }

    System.out.println("Сумма: " + sum);
  }
}
```

В результате запуска программа генерирует следующий вывод:

```
Значение: 1
Значение: 2
Значение: 3
Значение: 4
Значение: 5
Значение: 6
Значение: 7
Значение: 8
Значение: 9
Значение: 10
Сумма: 55
```

В выводе видно, что цикл `for` в стиле "for-each" автоматически последовательно проходит по массиву от наименьшего индекса до наибольшего.

Хотя цикл `for` в стиле "for-each" повторяется до тех пор, пока не будут просмотрены все элементы массива, цикл можно прервать досрочно, используя оператор `break`. Например, представленная далее программа суммирует только первые пять элементов числа:

```
// Использование break с циклом for в стиле "for-each".
class ForEach2 {
  public static void main(String[] args) {
    int sum = 0;
    int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    // Применить цикл for в стиле "for-each" для отображения
    // и суммирования значений.
    for(int x : nums) {
      System.out.println("Значение: " + x);
      sum += x;
      if(x == 5) break;   // остановить выполнение цикла,
                          // когда получено значение 5
    }
    System.out.println("Сумма первых пяти элементов: " + sum);
  }
}
```

Вот какой вывод будет получен:

```
Значение: 1
Значение: 2
Значение: 3
Значение: 4
Значение: 5
Сумма первых пяти элементов: 15
```

Легко заметить, что цикл `for` останавливается после получения пятого элемента. Оператор `break` можно также использовать с другими циклами Java, и он подробно обсуждается позже в главе.

Существует один аспект, касающийся цикла `for` в стиле "for-each", который важно понимать. Его переменная итерации доступна только для чтения, хотя она связана с лежащим в основе массивом. Присваивание значения переменной итерации не влияет на лежащий в основе массив. Другими словами, изменить содержимое массива, присваивая переменной итерации новое значение, не удастся.

Например, рассмотрим следующую программу:

```
// Переменная итерации цикла for в стиле "for-each" доступна только для чтения.
class NoChange {
  public static void main(String[] args) {
    int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    for(int x: nums) {
      System.out.print(x + " ");
      x = x * 10;  // не влияет на nums
    }

    System.out.println();

    for(int x : nums)
      System.out.print(x + " ");

    System.out.println();
  }
}
```

В первом цикле `for` значение переменной итерации увеличивается в 10 раз. Однако такое присваивание никак не влияет на лежащий в основе массив `nums`, что иллюстрирует второй цикл `for`. Сказанное подтверждается выводом программы:

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Проход по многомерным массивам

Расширенная версия `for` работает также и с многомерными массивами. Но не забывайте о том, что многомерные массивы в Java представляют собой *массивы массивов*. (Например, двумерный массив — это массив одномерных массивов.) Данный факт важен при проходе по многомерному массиву, поскольку на каждой итерации получается *очередной массив*, а не индивидуальный элемент. Кроме того, переменная итерации в цикле `for` должна быть совместимой с типом получаемого массива. Например, в случае двумерного массива переменная итерации должна быть ссылкой на одномерный массив. В общем случае при использовании цикла `for` в стиле "for-each" для прохода по массиву с N измерениями полученные объекты будут массивами с $N-1$ измерениями. Чтобы понять последствия, рассмотрим следующую программу. В ней применяются вложенные циклы `for` для получения элементов двумерного массива в порядке строк, от первой до последней.

```
// Использование цикла for в стиле "for-each" для прохода по двумерному массиву.
class ForEach3 {
  public static void main(String[] args) {
    int sum = 0;
    int[][] nums = new int[3][5];

    // Сохранить в nums ряд значений.
    for(int i = 0; i < 3; i++)
      for(int j = 0; j < 5; j++)
        nums[i][j] = (i+1)*(j+1);
```

```
    // Применить цикл for в стиле "for-each" для отображения
    // и суммирования значений.
    for(int[] x : nums) {
      for(int y : x) {
        System.out.println("Значение: " + y);
        sum += y;
      }
    }
    System.out.println("Сумма: " + sum);
  }
}
```

Ниже показан вывод, генерируемый программой:

```
Значение: 1
Значение: 2
Значение: 3
Значение: 4
Значение: 5
Значение: 2
Значение: 4
Значение: 6
Значение: 8
Значение: 10
Значение: 3
Значение: 6
Значение: 9
Значение: 12
Значение: 15
Сумма: 90
```

Вот строка программы, представляющая особый интерес:

```
for(int[] x: nums) {
```

Обратите внимание на то, как объявлена переменная `x`. Она является ссылкой на одномерный массив целых чисел. Необходимость в таком объявлении связана с тем, что на каждой итерации цикла `for` получается очередной *массив* из `nums`, начиная с массива `nums[0]`. Затем во внутреннем цикле `for` производится проход по каждому из этих массивов с отображением значения каждого элемента.

Применение расширенного цикла `for`

Поскольку цикл `for` в стиле "for-each" способен только последовательно проходить по массиву от начала до конца, вам может показаться, что его использование ограничено, но это не так. Большое количество алгоритмов требует именно такого механизма. Одним из самых распространенных можно считать поиск. Например, в следующей программе цикл `for` применяется для поиска значения в несортированном массиве. Он останавливается, когда значение найдено.

```
// Поиск в массиве с использованием цикла for в стиле "for-each".
class Search {
```

```
  public static void main(String[] args) {
    int[] nums = { 6, 8, 3, 7, 5, 6, 1, 4 };
    int val = 5;
    boolean found = false;

    // Применить цикл for в стиле "for-each" для поиска val в nums.
    for(int x : nums) {
      if(x == val) {
        found = true;
        break;
      }
    }

    if(found)
      System.out.println("Значение найдено!");
  }
}
```

Цикл `for` в стиле "for-each" — великолепный вариант в этом приложении, т.к. поиск в несортированном массиве предусматривает последовательный просмотр каждого элемента. (Разумеется, если бы массив был отсортирован, можно было бы использовать двоичный поиск, что потребовало бы цикл отличающегося стиля.) Другие типы приложений, которые выигрывают от циклов `for` в стиле "for-each", включают вычисление среднего значения, нахождение минимального или максимального значения внутри множества, поиск дубликатов и т.д.

Хотя в примерах, приводимых в главе, применялись массивы, цикл `for` в стиле "for-each" особенно полезен при работе с коллекциями, определенными в инфраструктуре Collections Framework, который описан в части II. В более общем случае цикл `for` может проходить по элементам любой коллекции объектов, пока она удовлетворяет набору ограничений, которые описаны в главе 20.

Выведение типов локальных переменных в цикле `for`

Как объяснялось в главе 3, в версии JDK 10 появилось средство, называемое *выведением типов локальных переменных*, которое позволяет определить тип локальной переменной по типу ее инициализатора. Для использования выведения типов локальных переменных понадобится указать `var` в качестве типа переменной и инициализировать ее. Выведение типов локальных переменных можно применять в цикле `for` при объявлении и инициализации переменной управления циклом внутри традиционного цикла `for` или при указании переменной итерации в цикле `for` в стиле "for-each". В следующей программе демонстрируются примеры каждого случая:

```
// Использование выведения типов локальных переменных в цикле for.
class TypeInferenceInFor {
  public static void main(String[] args) {

    // Применить выведение типов к переменной управления циклом.
    System.out.print("Значения x: ");
```

```
    for(var x = 2.5; x < 100.0; x = x * 2)
      System.out.print(x + " ");

    System.out.println();

    // Применить выведение типов к переменной итерации.
    int[] nums = { 1, 2, 3, 4, 5, 6};
    System.out.print("Значения в массиве nums: ");
    for(var v : nums)
      System.out.print(v + " ");

    System.out.println();
  }
}
```

Вот вывод программы:

```
Значения x: 2.5 5.0 10.0 20.0 40.0 80.0
Значения в массиве nums: 1 2 3 4 5 6
```

В приведенном примере для переменной управления циклом x в строке:

```
for(var x = 2.5; x < 100.0; x = x * 2)
```

выводится тип double по причине типа ее инициализатора. Для переменной итерации v в строке:

```
for(var v : nums)
```

выводится тип int, потому что он является типом элементов массива nums.

И последнее замечание: поскольку многие читатели будут работать в средах, предшествующих JDK 10, в большинстве циклов for в оставшихся материалах настоящего издания книги выведение типов локальных переменных использоваться не будет. Конечно, вы должны обдумать его применение в новом коде, который вам предстоит написать.

Вложенные циклы

Подобно всем остальным языкам программирования в Java разрешены вложенные циклы, т.е. один цикл может находиться внутри другого. Например, в показанной ниже программе используются вложенные циклы for:

```
// Циклы могут быть вложенными.
class Nested {
  public static void main(String[] args) {
    int i, j;

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

Программа генерирует следующий вывод:

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . .
. . . .
. . .
. .
.
```

Операторы перехода

В Java поддерживаются три оператора перехода: `break`, `continue` и `return`. Они передают управление другой части программы и обсуждаются ниже.

На заметку! Помимо обсуждаемых здесь операторов перехода в Java есть еще один способ, с помощью которого можно изменить поток выполнения программы: через обработку исключений. Обработка исключений обеспечивает структурированный подход, с помощью которого программа может перехватывать и обрабатывать ошибки во время выполнения. Он поддерживается ключевыми словами `try`, `catch`, `throw`, `throws` и `finally`. По существу механизм обработки исключений позволяет программе выполнять нелокальный переход. Поскольку обработка исключений является обширной темой, она обсуждается отдельно в главе 10.

Использование оператора `break`

Оператор `break` в языке Java применяется в трех ситуациях. Во-первых, как вы видели, он завершает последовательность операторов в операторе `switch`. Во-вторых, его можно использовать для выхода из цикла. В-третьих, его можно применять как "цивилизованную" форму перехода в стиле "goto". Здесь объясняются последние два использования.

Использование оператора `break` для выхода из цикла

За счет применения оператора `break` вы можете принудительно завершить цикл, пропуская вычисление условного выражения и выполнение любого оставшегося кода в теле цикла. Когда внутри цикла встречается оператор `break`, цикл завершается и управление передается оператору, следующему за циклом. Вот простой пример:

```
// Использование break для выхода из цикла.
class BreakLoop {
  public static void main(String[] args) {
    for(int i=0; i<100; i++) {
      if(i == 10) break;  // прекратить выполнение цикла, если i равно 10
      System.out.println("i: " + i);
    }
    System.out.println("Цикл завершен.");
  }
}
```

Программа генерирует следующий вывод:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Цикл завершен.
```

Несложно заметить, что хотя цикл for предназначен для выполнения от 0 до 99, оператор break приводит к его преждевременному прекращению, когда i равно 10.

Оператор break можно использовать с любыми циклами Java, включая намеренно реализованные бесконечные циклы. Например, ниже показана предыдущая программа, переписанная с применением цикла while. Вывод этой программы будет таким же, как приведенный выше.

```
// Использование break для выхода из цикла while.
class BreakLoop2 {
  public static void main(String[] args) {
    int i = 0;

    while(i < 100) {
      if(i == 10) break;  // прекратить выполнение цикла, если i равно 10
      System.out.println("i: " + i);
      i++;
    }
    System.out.println("Цикл завершен.");
  }
}
```

При использовании внутри набора вложенных циклов оператор break производит выход только из самого внутреннего цикла, например:

```
// Использование break с вложенными циклами.
class BreakLoop3 {
  public static void main(String[] args) {
    for(int i=0; i<3; i++) {
      System.out.print("Проход " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break;  // прекратить выполнение цикла, если j равно 10
        System.out.print(j + " ");
      }
      System.out.println();
    }
    System.out.println("Цикл завершен.");
  }
}
```

Программа генерирует следующий вывод:

```
Проход 0: 0 1 2 3 4 5 6 7 8 9
Проход 1: 0 1 2 3 4 5 6 7 8 9
Проход 2: 0 1 2 3 4 5 6 7 8 9
Цикл завершен.
```

Как видите, оператор `break` во внутреннем цикле приводит к прекращению только этого цикла, не затрагивая внешний цикл.

С оператором `break` связаны еще два момента, о которых следует помнить. Во-первых, в цикле могут находиться более одного оператора `break`. Однако будьте осторожны. Слишком большое количество операторов `break` способно деструктурировать код. Во-вторых, оператор `break`, завершающий `switch`, влияет только на этот оператор `switch`, но не на любые объемлющие циклы.

Помните! Оператор `break` не задумывался как обычное средство завершения цикла. Для этой цели предназначено условное выражение цикла. Оператор `break` должен применяться для прекращения работы цикла только в случае возникновения какой-то особой ситуации.

Использование оператора `break` как разновидности перехода в стиле "goto"

В дополнение к применению с оператором `switch` и циклами оператор `break` также может использоваться сам по себе, чтобы обеспечить "цивилизованную" форму перехода в стиле "goto". В языке Java нет оператора "goto", т.к. он обеспечивает возможность ветвления произвольным и неструктурированным образом, что обычно затрудняет понимание и сопровождение кода, опирающегося на переходы в стиле "goto". Кроме того, "goto" препятствует некоторым оптимизациям со стороны компилятора. Тем не менее, есть несколько мест, где переходы в стиле "goto" будут ценной и законной конструкцией для управления потоком. Например, переход в стиле "goto" может быть полезен при выходе из глубоко вложенных циклов. Для обработки таких ситуаций в Java определена расширенная форма оператора `break`. С применением такой формы `break` можно, например, выходить из одного или нескольких блоков кода, которые не обязательно должны являться частью цикла или переключателя, а могут быть любыми. Более того, можно точно указывать, где будет возобновлено выполнение, т.к. расширенная форма оператора `break` работает с меткой. Как вы увидите, `break` обеспечивает преимущества перехода в стиле "goto" без присущих ему проблем.

Общая форма оператора `break` с меткой выглядит следующим образом:

```
break метка;
```

Чаще всего `метка` представляет собой имя маркера, идентифицирующего блок кода. Блок кода может быть как автономным, так и блоком, являющимся целью другого оператора. При выполнении расширенной формы оператора `break` поток управления покидает блок, указанный в `break`. Снабженный меткой блок должен охватывать оператор `break`, но не обязательно быть тем, который содержит в себе этот `break` непосредственно. Отсюда следует, на-

пример, что оператор break с меткой можно использовать для выхода из набора вложенных блоков. Но применять break для передачи управления из блока, который не охватывает данный оператор break, нельзя.

Чтобы назначить блоку имя, необходимо поместить в его начало метку. *Метка* — это любой допустимый идентификатор Java, за которым следует двоеточие. После пометки блока метку можно использовать в качестве цели оператора break, что приведет к возобновлению выполнения после *конца* помеченного блока. Например, в показанной далее программе реализованы три вложенных блока, каждый со своей меткой. Оператор break передает управление вперед, за конец блока с меткой second, пропуская два оператора println().

```
// Использование break в качестве "цивилизованной" формы перехода в стиле "goto"
class Break {
  public static void main(String[] args) {
    boolean t = true;

    first: {
      second: {
        third: {
          System.out.println("Перед оператором break.");
          if(t) break second;  // выйти из блока second
          System.out.println("Этот оператор не выполнится.");
        }
        System.out.println("Этот оператор не выполнится.");
      }
      System.out.println("После блока second.");
    }
  }
}
```

В результате запуска программы получается следующий вывод:

```
Перед оператором break.
После блока second.
```

Одним из наиболее распространенных применений оператора break с меткой является выход из вложенных циклов. Скажем, в приведенной ниже программе внешний цикл выполняется только один раз:

```
// Использование break для выхода из вложенных циклов.
class BreakLoop4 {
  public static void main(String[] args) {
    outer: for(int i=0; i<3; i++) {
      System.out.print("Проход " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break outer;  // выйти из обоих циклов
        System.out.print(j + " ");
      }
      System.out.println("Это выводиться не будет.");
    }
    System.out.println("Циклы завершены.");
  }
}
```

Вот вывод, генерируемый программой:

```
Проход 0: 0 1 2 3 4 5 6 7 8 9 Циклы завершены.
```

После выхода из внутреннего цикла во внешний цикл оба цикла завершаются. Обратите внимание на то, что здесь помечен оператор `for`, содержащий целевой блок кода.

Имейте в виду, что использовать оператор `break` с меткой, которая определена не для охватывающего блока, не разрешено. Например, следующая программа содержит ошибку и компилироваться не будет:

```
// Эта программа содержит ошибку.
class BreakErr {
  public static void main(String[] args) {

    one: for(int i=0; i<3; i++) {
      System.out.print("Проход " + i + ": ");
    }

    for(int j=0; j<100; j++) {
      if(j == 10) break one;  // ОШИБКА
      System.out.print(j + " ");
    }
  }
}
```

Поскольку цикл, помеченный как `one`, не охватывает оператор `break`, передать управление из этого блока невозможно.

Использование оператора `continue`

Иногда необходимо обеспечить, чтобы итерация цикла выполнилась раньше, до достижения конца тела. То есть выполнение цикла должно продолжаться, но без обработки остатка кода в его теле для конкретной итерации. По сути, это переход в конец цикла. Такое действие реализует оператор `continue`. В циклах `while` и `do-while` оператор `continue` передает управление напрямую условному выражению, управляющему циклом. В цикле `for` управление передается сначала итерационной части оператора `for`, а затем условному выражению. Для всех трех циклов любой промежуточный код пропускается.

В показанной далее программе оператор `continue` применяется для вывода в каждой строке двух чисел:

```
// Демонстрация работы continue.
class Continue {
  public static void main(String[] args) {
    for(int i=0; i<10; i++) {
      System.out.print(i + " ");
      if (i%2 == 0) continue;
      System.out.println("");
    }
  }
}
```

Операция % в коде используется для проверки значения i на предмет четности. Если значение i четное, тогда цикл продолжается без вывода символа новой строки.

Программа генерирует следующий вывод:

```
0 1
2 3
4 5
6 7
8 9
```

Как и в случае с break, в операторе continue можно указывать метку для описания того, какой объемлющий цикл необходимо продолжить. Вот пример программы, в которой оператор continue применяется для вывода треугольной таблицы умножения чисел от 0 до 9:

```
// Использование continue с меткой.
class ContinueLabel {
  public static void main(String[] args) {
    outer: for (int i=0; i<10; i++) {
      for(int j=0; j<10; j++) {
        if(j > i) {
          System.out.println();
          continue outer;
        }
        System.out.print(" " + (i * j));
      }
    }
    System.out.println();
  }
}
```

Оператор continue в этом примере завершает цикл по j и продолжает со следующей итерации цикла по i. Вывод программы показан ниже:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

Подходящие сценарии использования continue встречаются редко. Одна из причин связана с тем, что язык Java предлагает обширный набор операторов циклов, которые подходят для большинства приложений. Однако для тех особых обстоятельств, когда итерацию необходимо начинать раньше, оператор continue обеспечивает структурированный способ решения задачи.

Оператор return

Последний управляющий оператор — return. Он применяется для явного возвращения из метода, т.е. управление программой передается обратно вызывающей стороне. Таким образом, return классифицируется как оператор перехода. Хотя подробное обсуждение оператора return следует отложить до обсуждения методов в главе 6, здесь представлен его краткий обзор.

Оператор return можно использовать в любом месте метода, чтобы возвратить управление вызывающей стороне. Таким образом, оператор return немедленно завершает выполнение метода, в котором он находится. Сказанное иллюстрируется в следующем примере, где оператор return возвращает управление исполняющей среде Java, поскольку именно она вызвала main():

```
// Демонстрация работы return.
class Return {
  public static void main(String[] args) {
    boolean t = true;

    System.out.println("Перед оператором return.");
    if(t) return;     // возвратить управление вызывающей стороне
    System.out.println("Это выполняться не будет.");
  }
}
```

Вот как выглядит вывод программы:

```
Перед оператором return.
```

Как видите, последний оператор println() не выполняется. Сразу после выполнения оператора return управление возвращается вызывающей стороне.

И последнее замечание: в предыдущей программе оператор if(t) обязателен. Без него компилятор Java отметил бы ошибку типа "недостижимый код" (unreachable code), поскольку компилятору было бы известно, что последний оператор println() никогда не выполнится. Для предотвращения такой ошибки в коде и применяется оператор if, чтобы "обмануть" компилятор ради этой демонстрации.

ГЛАВА

6 Введение в классы

Класс лежит в самом центре Java. Он представляет собой логическую конструкцию, на которой построен весь язык Java, потому что она определяет форму и природу объекта. Таким образом, класс формирует основу для объектно-ориентированного программирования (ООП) на Java. Любая концепция, которую вы хотите реализовать в программе на Java, должна быть инкапсулирована внутри класса.

Поскольку класс настолько фундаментален для Java, ему будут посвящены эта и несколько последующих глав. Здесь вы ознакомитесь с основными элементами класса и узнаете, как использовать класс для создания объектов. Вы также узнаете о методах, конструкторах и ключевом слове `this`.

Основы классов

Классы применяются с самого начала книги. Однако до сих пор была показана только самая рудиментарная форма класса. Классы, создаваемые в предшествующих главах, главным образом существовали просто для инкапсуляции метода `main()`, который использовался с целью демонстрации основ синтаксиса Java. Как вы увидите, классы значительно мощнее, чем представленные до сих пор ограниченные варианты.

Вероятно, наиболее важная характеристика класса заключается в том, что он определяет новый тип данных. После определения новый тип можно применять для создания объектов такого типа. Следовательно, класс — это *шаблон* для объекта, а объект — это *экземпляр* класса. Так как объект является экземпляром класса, вы часто будете видеть, что слова *объект* и *экземпляр* используются взаимозаменяемо.

Общая форма класса

При определении класса вы объявляете его точную форму и природу, для чего указываете данные, которые он содержит, и код, который работает с этими данными. В то время как очень простые классы могут содержать только код или только данные, большинство реальных классов содержат то и другое. Как вы увидите, код класса определяет интерфейс к своим данным.

Класс объявляется с применением ключевого слова `class`. Классы, которые использовались до сих пор, на самом деле являются крайне ограниченными примерами его полной формы. Классы могут (и обычно становятся) намного сложнее. Ниже приведена упрощенная общая форма определения класса:

```
class имя-класса {
  тип переменная-экземпляра1;
  тип переменная-экземпляра2;
  // ...
  тип переменная-экземпляраN;

  тип имя-метода1(список-параметров) {
    // тело метода
  }
  тип имя-метода2(список-параметров) {
    // тело метода
  }
  // ...
  тип имя-методаN(список-параметров) {
    // тело метода
  }
}
```

Данные, или переменные, определенные в классе, называются *переменными экземпляра*. Код содержится внутри методов. В совокупности методы и переменные, определенные в классе, называются *членами* класса. В большинстве классов переменные экземпляра обрабатываются и доступны с помощью методов, определенных для этого класса. Таким образом, как правило, именно методы определяют, как можно использовать данные класса.

Переменные, определенные внутри класса, называются переменными экземпляра из-за того, что каждый экземпляр класса (т.е. каждый объект класса) содержит собственную копию этих переменных. Таким образом, данные для одного объекта являются уникальными и обособленными от данных для другого. Вскоре мы продолжим раскрытие темы, но о настолько важной концепции стоит узнать пораньше.

Все методы имеют ту же общую форму, что и метод `main()`, который мы применяли до сих пор, но большинство методов не будут указаны как `static` или `public`. Обратите внимание, что в общей форме класса не определен метод `main()`. Классы Java не обязаны иметь метод `main()`. Он указывается только в том случае, если класс является начальной точкой для выполнения программы. Кроме того, некоторые виды приложений Java вообще не требуют метода `main()`.

Простой класс

Давайте начнем изучение классов с простого примера. Ниже приведен код класса `Box` (коробка), в котором определены три переменных экземпляра: `width` (ширина), `height` (высота) и `depth` (глубина). Пока что `Box` не содержит какие-либо методы (но со временем они будут добавляться).

```
class Box {
  double width;
  double height;
  double depth;
}
```

Как утверждалось выше, класс определяет новый тип данных. В рассматриваемом случае новый тип данных называется Box. Имя класса будет использоваться для объявления объектов типа Box. Важно помнить, что объявление класса создает только шаблон, но не фактический объект. Таким образом, предыдущий код не создает никаких объектов типа Box.

Чтобы действительно создать объект Box, будет применяться оператор следующего вида:

```
Box mybox = new Box(); // создать объект Box по имени mybox
```

После выполнения этого оператора mybox будет ссылаться на экземпляр Box. Таким образом, он обретет "физическую" реальность. Пока что не беспокойтесь о деталях представленного оператора.

Как упоминалось ранее, при создании экземпляра класса на самом деле создается объект, который содержит собственную копию каждой переменной экземпляра, определенной в классе. Соответственно каждый объект Box будет содержать собственные копии переменных экземпляра width, height и depth. Для доступа к этим переменным будет использоваться операция *точки* (.), которая связывает имя переменной экземпляра с именем объекта. Например, чтобы присвоить переменной width объекта mybox значение 100, потребуется применить такой оператор:

```
mybox.width = 100;
```

Приведенный оператор сообщает компилятору о необходимости присвоить копии width, содержащейся в объекте mybox, значение 100. В общем случае операция точки используется для доступа и к переменным экземпляра, и к методам внутри объекта. Еще один момент: несмотря на то, что ее обычно называют *операцией* точки, формальная спецификация Java классифицирует . как разделитель. Тем не менее, из-за широкого распространения термина "операция точки" именно он и применяется в книге.

Ниже показана полная программа, в которой используется класс Box:

```
/* Программа, в которой используется класс Box.
   Назовите этот файл BoxDemo.java
*/
class Box {
  double width;
  double height;
  double depth;
}
// В этом классе объявляется объект типа Box.
class BoxDemo {
  public static void main(String[] args) {
    Box mybox = new Box();
    double vol;
```

```
    // Присвоить значения переменным экземпляра mybox.
    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;

    // Вычислить объем коробки.
    vol = mybox.width * mybox.height * mybox.depth;

    System.out.println("Объем равен " + vol);
  }
}
```

Вы должны назначить файлу, содержащему эту программу, имя BoxDemo.java, потому что метод main() находится в классе BoxDemo, а не в Box. Скомпилировав программу, вы обнаружите, что были созданы два файла .class — один для Box и один для BoxDemo. Компилятор Java автоматически помещает каждый класс в отдельный файл .class. Классы Box и BoxDemo не обязательно должны находиться в одном и том же исходном файле. Вы можете поместить классы в собственные файлы с именами Box.java и BoxDemo.java.

Для запуска программы потребуется выполнить BoxDemo.class. В результате будет получен следующий вывод:

```
Объем равен 3000.0
```

Как утверждалось ранее, каждый объект имеет собственные копии переменных экземпляра, т.е. если есть два объекта Box, то каждый из них имеет собственную копию depth, width и height. Важно понимать, что изменения в переменных экземпляра одного объекта не влияют на переменные экземпляра другого. Например, в следующей программе объявляются два объекта Box:

```
// В этой программе объявляются два объекта Box.
class Box {
  double width;
  double height;
  double depth;
}
class BoxDemo2 {
  public static void main(String[] args) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // Присвоить значения переменным экземпляра mybox1.
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* Присвоить переменным экземпляра mybox2 другие значения. */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // Вычислить объем первой коробки.
    vol = mybox1.width * mybox1.height * mybox1.depth;
    System.out.println("Объем равен " + vol);
```

```
    // Вычислить объем второй коробки.
    vol = mybox2.width * mybox2.height * mybox2.depth;
    System.out.println("Объем равен " + vol);
  }
}
```

Вот вывод, который генерирует программа:

```
Объем равен 3000.0
Объем равен 162.0
```

Вы видите, что данные в mybox1 полностью независимы от данных, содержащихся в mybox2.

Объявление объектов

Как только что объяснялось, создание класса означает создание нового типа данных, который можно применять для объявления объектов этого типа. Однако получение объектов класса представляет собой двухэтапный процесс. Во-первых, потребуется объявить переменную типа класса. Такая переменная не определяет объект, а просто может *ссылаться* на объект. Во-вторых, необходимо получить физическую копию объекта и присвоить ее этой переменной, для чего служит операция new. Операция new динамически (т.е. во время выполнения) выделяет память для объекта и возвращает ссылку на нее, которая по существу является адресом в памяти объекта, выделенной new. Затем ссылка сохраняется в переменной. Таким образом, в Java все объекты класса должны размещаться динамически. Рассмотрим детали данной процедуры. В предшествующих примерах программ для объявления объекта типа Box использовалась строка следующего вида:

```
Box mybox = new Box();
```

Показанный оператор объединяет два только что описанных шага. Его можно переписать так, чтобы более четко показать каждый шаг:

```
Box mybox;          // объявить ссылку на объект
mybox = new Box();  // разместить в памяти объект Box
```

В первой строке переменная mybox объявляется как ссылка на объект типа Box. Пока что mybox не ссылается на фактический объект. Во второй строке объекта размещается в памяти и ссылка на него присваивается mybox. После выполнения второй строки переменную mybox можно использовать, как если бы она была объектом Box. Но на самом деле mybox просто содержит адрес памяти фактического объекта Box. Действие приведенных выше двух строк кода иллюстрируется на рис. 6.1.

Подробный анализ операции new

Как только что объяснялось, операция new динамически выделяет память для объекта. В контексте присваивания она имеет следующую общую форму:

```
переменная-класса = new имя-класса();
```

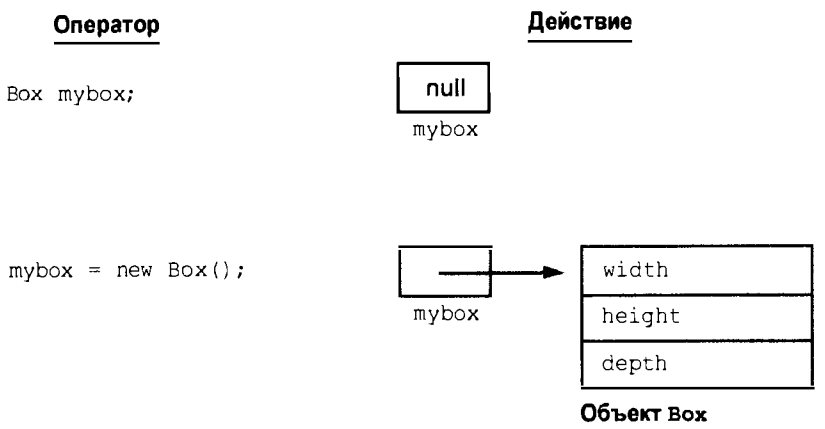


Рис. 6.1. Объявление объекта типа Box

Здесь *переменная-класса* — это переменная создаваемого типа класса, а *имя-класса* — это имя класса, экземпляр которого создается. Имя класса, за которым следуют круглые скобки, указывает *конструктор* класса. Конструктор определяет, что происходит при создании объекта класса. Конструкторы являются важной частью всех классов и имеют много значимых атрибутов. В большинстве реальных классов конструкторы определяются явно внутри определений классов. Тем не менее, если явный конструктор не указан, то компилятор Java автоматически предоставит стандартный конструктор, что и происходило в случае с Box. Пока что мы будем применять стандартный конструктор. Вскоре вы увидите, каким образом определять собственные конструкторы.

На данном этапе вас может интересовать, почему не нужно использовать new для таких вещей, как целые числа или символы. Дело в том, что примитивные типы Java реализованы не в виде объектов, а в форме "обычных" переменных. Так поступили в интересах эффективности. Как вы увидите, объекты обладают многими характеристиками, которые требуют, чтобы их трактовали иначе, чем примитивные типы. Отсутствие в примитивных типах накладных расходов, присущих объектам, дало возможность реализовать примитивные типы более эффективно. Позже вы встретите объектные версии примитивных типов, доступные для использования в тех ситуациях, когда требуются полноценные объекты таких типов.

Важно понимать, что операция new выделяет память для объекта во время выполнения. Преимущество этого подхода состоит в том, что ваша программа может создать столько объектов, сколько требуется во время ее выполнения. Однако поскольку память конечна, возможно, что new не сможет выделить память под объект из-за нехватки памяти. В такой ситуации возникает исключение времени выполнения. (Вы узнаете, как обрабатывать исключения, в главе 10.) В примерах программ, приводимых в книге, вам не придется беспокоиться о нехватке памяти, но нужно будет учитывать эту возможность в реальных программах, которые вы будете писать.

Давайте еще раз подчеркнем различие между классом и объектом. Класс создает новый тип данных, который можно применять для создания объектов. То есть класс создает логическую инфраструктуру, определяющую отношения между его членами. При объявлении объекта класса создается экземпляр этого класса. Таким образом, класс является логической конструкцией, а объект имеет физическую реальность (занимает место в памяти). Важно четко осознавать указанное различие.

Присваивание для переменных ссылок на объекты

При присваивании переменные ссылок на объекты действуют иначе, чем вы могли бы ожидать. Например, что, по-вашему, делает следующий фрагмент кода?

```
Box b1 = new Box();
Box b2 = b1;
```

Вы можете подумать, что переменной b2 присваивается ссылка на копию объекта, на который ссылается b1. То есть вы могли бы предположить, что b1 и b2 относятся к обособленным объектам. Тем не менее, вы бы ошибались. Взамен после выполнения данного фрагмента кода переменные b1 и b2 будут ссылаться на *тот же самый* объект. Присваивание переменной b2 значения b1 не привело к выделению памяти или копированию какой-либо части исходного объекта. Оно просто заставляет b2 ссылаться на тот же объект, что и b1. Таким образом, любые изменения, внесенные в объект через переменную b2, повлияют на объект, на который ссылается b1, поскольку это один и тот же объект. Ситуация иллюстрируется на рис. 6.2.

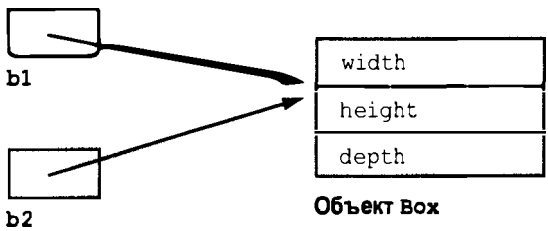


Рис. 6.2. Присваивание переменных ссылок на объекты

Хотя переменные b1 и b2 относятся к тому же самому объекту, они никак не связаны друг с другом. Например, последующее присваивание переменной b1 значения null просто *отсоединит* b1 от исходного объекта, не затрагивая объект и переменную b2:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Здесь b1 устанавливается в null, но b2 по-прежнему указывает на исходный объект.

Помните! Когда вы присваиваете значение одной переменной ссылки на объект другой переменной ссылки на объект, копия объекта не создается, а создается только копия ссылки.

Введение в методы

В начале главы упоминалось, что классы обычно состоят из двух вещей: переменных экземпляра и методов. Тема методов обширна, потому что Java наделяет их широкими возможностями и гибкостью. Фактически методам посвящена большая часть следующей главы. Однако есть определенные основы, которые необходимо усвоить уже сейчас, чтобы приступить к добавлению методов в свои классы.

Вот общая форма метода:

```
тип имя(список-параметров) {
  // тело метода
}
```

Здесь в *тип* указывается тип данных, возвращаемых методом. Он может быть любым допустимым типом, включая создаваемые вами типы классов. Если метод не возвращает значение, то его возвращаемым типом должен быть void. Имя метода указывается в имя. Это может быть любой законный идентификатор кроме тех, которые уже используются другими элементами в текущей области видимости. Наконец, *список-параметров* представляет собой последовательность пар типов и идентификаторов, разделенных запятыми. Параметры, по сути, являются переменными, которые получают значение аргументов, переданных методу при его вызове. Если у метода нет параметров, то список параметров будет пустым.

Методы, которые имеют тип возвращаемого значения, отличающийся от void, возвращают значение вызывающей процедуре с применением следующей формы оператора return:

```
return значение;
```

Здесь *значение* указывает возвращаемое значение.

В последующих нескольких разделах вы увидите, как создавать различные типы методов, в том числе те, которые принимают параметры, и те, которые возвращают значения.

Добавление метода в класс Box

Хотя совершенно нормально создавать класс, содержащий только данные, такое случается редко. Большую часть времени вы будете использовать методы для доступа к переменным экземпляра, которые определены классом. В действительности методы определяют интерфейс для большинства классов, что позволяет разработчику класса скрыть конкретную реализацию вну-

тренних структур данных за более привлекательными абстракциями методов. Помимо определения методов, обеспечивающих доступ к данным, вы также можете определять методы, которые применяются внутри самого класса.

Давайте начнем с добавления метода в класс Box. При просмотре предшествующих программ вам могло прийти в голову, что вычисление объема коробки лучше выполнять классом Box, а не классом BoxDemo. В конце концов, поскольку объем коробки зависит от размеров коробки, для его вычисления имеет смысл использовать класс Box. Тогда в класс Box потребуется добавить метод, как показано ниже:

```
// В этой программе внутрь класса Box добавляется метод.
class Box {
  double width;
  double height;
  double depth;

  // Отобразить объем коробки.
  void volume() {
    System.out.print("Объем равен ");
    System.out.println(width * height * depth);
  }
}

class BoxDemo3 {
  public static void main(String[] args) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    // Присвоить значения переменным экземпляра mybox1.
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* Присвоить переменным экземпляра mybox2 другие значения. */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // Отобразить объем первой коробки.
    mybox1.volume();

    // Отобразить объем второй коробки.
    mybox2.volume();
  }
}
```

Программа генерирует такой же вывод, как и ее предыдущая версия:

```
Объем равен 3000.0
Объем равен 162.0
```

Внимательно взгляните на следующие две строки кода:

```
mybox1.volume();
mybox2.volume();
```

В первой строке вызывается метод volume() на mybox1, т.е. volume() вызывается в отношении объекта mybox1 с применением имени объекта и опе-

рации точки. Таким образом, вызов `mybox1.volume()` отображает объем коробки, определенной переменной `mybox1`, а вызов `mybox2.volume()` — объем коробки, определенной переменной `mybox2`. Каждый раз, когда вызывается метод `volume()`, он отображает объем указанной коробки.

Если вы не знакомы с концепцией вызова метода, то следующее обсуждение поможет прояснить ситуацию. При выполнении вызова `mybox1.volume()` исполняющая среда Java передает управление коду, определенному внутри `volume()`. После выполнения инструкций метода `volume()` управление возвращается вызывающей процедуре и выполнение возобновляется со строки кода, следующей за вызовом. В самом общем смысле метод представляет собой способ реализации подпрограмм в Java.

В методе `volume()` имеется кое-что очень важное, на что следует обратить внимание: ссылка на переменные экземпляра `width`, `height` и `depth` производится напрямую, без предшествующего имени объекта или операции точки. Когда метод использует переменную экземпляра, определенную его классом, он делает это напрямую, без явной ссылки на объект и без применения операции точки. Если подумать, то причину понять легко. Метод всегда вызывается в отношении некоторого объекта своего класса. После того как вызов произошел, объект становится известным. Таким образом, внутри метода нет необходимости указывать объект во второй раз. Это означает, что `width`, `height` и `depth` внутри `volume()` неявно относятся к копиям переменных, которые находятся в объекте, на котором вызван метод `volume()`.

Итак, еще раз: когда доступ к переменной экземпляра осуществляется кодом, который не входит в состав класса, где определена данная переменная экземпляра, то его придется делать через объект с использованием операции точки. Тем не менее, когда к переменной экземпляра обращается код, который входит в состав того же класса, что и переменная экземпляра, то на переменную можно ссылаться напрямую. То же самое относится и к методам.

Возвращение значения

Хотя реализация `volume()` переносит вычисление объема коробки внутрь класса `Box`, такой способ нельзя считать наилучшим. Скажем, а что, если в другой части программы нужно узнать объем коробки, но не отображать его значение? Более удачный подход к реализации `volume()` предусматривает вычисление объема коробки и возвращение результата вызывающей стороне. Именно так сделано в показанной ниже улучшенной версии предыдущей программы:

```
// Теперь volume() возвращает объем коробки.

class Box {
  double width;
  double height;
  double depth;

  // Вычислить и возвратить объем.
  double volume() {
```

```
    return width * height * depth;
  }
}
class BoxDemo4 {
  public static void main(String[] args) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // Присвоить значения переменным экземпляра mybox1.
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* Присвоить переменным экземпляра mybox2 другие значения. */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // Получить объем первой коробки.
    vol = mybox1.volume();
    System.out.println("Объем равен " + vol);

    // Получить объем второй коробки.
    vol = mybox2.volume();
    System.out.println("Объем равен " + vol);
  }
}
```

Как видите, когда метод `volume()` вызывается, он помещается в правую часть оператора присваивания. Слева находится переменная, в данном случае `vol`, которая получит значение, возвращаемое `volume()`. Таким образом, после выполнения оператора:

```
vol = mybox1.volume();
```

значением `mybox1.volume()` будет `3000`, которое и сохранится в `vol`.

Касательно возвращаемых значений важно понимать два момента.

- Тип данных, возвращаемых методом, должен быть совместим с возвращаемым типом, который указан в методе. Например, если возвращаемым типом какого-то метода является `boolean`, то возвратить целое число не удастся.
- Тип переменной, которая получает значение, возвращаемое методом (`vol` в данном случае), тоже должен быть совместим с возвращаемым типом, указанным для метода.

И еще одно замечание: предыдущую программу можно написать немного эффективнее, потому что фактически она не нуждается в переменной `vol`. Вызов `volume()` можно было бы поместить непосредственно в оператор `println()`:

```
System.out.println("Объем равен " + mybox1.volume());
```

В таком случае при выполнении `println()` метод `mybox1.volume()` вызывается автоматически, а его значение передается в `println()`.

Добавление метода, принимающего параметры

В то время как некоторым методам параметры не нужны, большинству методов они необходимы. Параметры позволяют обобщить метод, т.е. параметризованный метод может работать с разнообразными данными и/или применяться в ряде ситуаций, немного отличающихся друг от друга. Чтобы проиллюстрировать сказанное, давайте воспользуемся очень простым примером. Ниже приведен метод, который возвращает квадрат числа 10:

```
int square()
{
  return 10 * 10;
}
```

Несмотря на то что метод действительно возвращает значение 10 в квадрате, его применение крайне ограничено. Однако если вы измените метод `square()` так, чтобы он принимал параметр, как показано далее, то сделаете его гораздо более полезным:

```
int square(int i)
{
  return i * i;
}
```

Теперь метод `square()` будет возвращать квадрат любого значения, с которым вызывается. Таким образом, `square()` становится методом общего назначения, который способен вычислять квадрат любого целочисленного значения, а не только 10. Вот пример:

```
int x, y;
x = square(5);  // x равно 25
x = square(9);  // x равно 81
y = 2;
x = square(y);  // x равно 4
```

В первом вызове `square()` параметру `i` передается значение 5. Во втором вызове `i` получает значение 9. В третьем вызове `i` передается значение переменной `y`, которое в данном примере равно 2. Как демонстрируется в примерах, метод `square()` может возвращать квадрат любых переданных значений.

Важно понимать различие между двумя терминами — параметр и аргумент. *Параметр* — это переменная, определенная методом, которая получает значение при вызове метода. Например, в методе `square()` параметром является `i`. *Аргумент* — это значение, которое передается методу при его вызове. Например, вызов `square(100)` передает в качестве аргумента значение `100`, которое получает параметр `i` внутри `square()`.

С помощью параметризованного метода класс `Box` можно усовершенствовать. В предшествующих примерах размеры каждой коробки должны были устанавливаться по отдельности с использованием последовательности операторов следующего вида:

```
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

Несмотря на то что код работает, он является источником беспокойства по двум причинам. Во-первых, он выглядит неуклюже и подвержен ошибкам. Скажем, довольно легко забыть установить какой-то размер. Во-вторых, в хорошо спроектированных программах на Java доступ к переменным экземпляра должен осуществляться только через методы, определенные их классом. В будущем вы сможете изменить поведение метода, но не сумеете изменить поведение открытой переменной экземпляра.

Таким образом, более эффективный подход к установке размеров коробки предусматривает создание метода, который принимает размеры коробки в своих параметрах и соответствующим образом устанавливает каждую переменную экземпляра. Данная концепция реализована в приведенной далее программе:

```
// В этой программе используется параметризованный метод.
class Box {
  double width;
  double height;
  double depth;

  // Вычислить и возвратить объем.
  double volume() {
    return width * height * depth;
  }

  // Установить размеры коробки.
  void setDim(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
}

class BoxDemo5 {
  public static void main(String[] args) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // Инициализировать объекты коробок.
    mybox1.setDim(10, 20, 15);
    mybox2.setDim(3, 6, 9);

    // Получить объем первой коробки.
    vol = mybox1.volume();
    System.out.println("Объем равен " + vol);

    // Получить объем второй коробки.
    vol = mybox2.volume();
    System.out.println("Объем равен " + vol);
  }
}
```

Как видите, для установки размеров каждого блока применяется метод `setDim()`.

Например, когда выполняется следующий вызов:

```
mybox1.setDim(10, 20, 15);
```

в параметр w копируется значение 10, в параметр h — значение 20, а в параметр d — значение 15. Затем внутри метода setDim() значения w, h и d присваиваются соответственно width, height и depth.

Многим читателям будут знакомы концепции, представленные в предшествующих разделах. Тем не менее, если такие понятия, как вызовы методов, аргументы и параметры, являются для вас новыми, тогда имеет смысл выделить некоторое время на экспериментирование с ними, прежде чем двигаться дальше. Концепции вызова методов, параметров и возвращаемых значений являются фундаментальными в программировании на Java.

Конструкторы

Инициализировать все переменные в классе при каждом создании его экземпляра может быть утомительно. Даже когда вы добавляете удобные методы вроде setDim(), было бы проще и лаконичнее выполнять всю настройку во время первоначального создания объекта. Поскольку требования к инициализации настолько распространены, в Java объектам разрешено инициализировать себя во время создания. Такая автоматическая инициализация выполняется с помощью конструктора.

Конструктор инициализирует объект немедленно после создания. Он имеет такое же имя, как у класса, где находится, и синтаксически похож на метод. После определения конструктор автоматически вызывается при создании объекта до завершения операции new. Конструкторы выглядят немного странно, потому что у них нет возвращаемого типа, даже void. Причина в том, что неявным возвращаемым типом конструктора класса является сам класс. Задача конструктора — инициализировать внутреннее состояние объекта, чтобы код, создающий экземпляр, немедленно получил в свое распоряжение полностью инициализированный и пригодный для использования объект.

Класс Box можно переделать, чтобы размеры коробки автоматически инициализировались при конструировании объекта, заменив метод setDim() конструктором. Начнем с определения простого конструктора, который устанавливает одинаковые значения для размеров каждой коробки. Вот как выглядит такая версия:

```
/* Здесь в Box используется конструктор
   для инициализации размеров коробки.
*/
class Box {
  double width;
  double height;
  double depth;

  // Это конструктор для Box.
  Box() {
```

```
    System.out.println("Конструирование Box");
    width = 10;
    height = 10;
    depth = 10;
  }

  // Вычислить и возвратить объем.
  double volume() {
    return width * height * depth;
  }
}

class BoxDemo6 {
  public static void main(String[] args) {
    // Объявить, разместить в памяти и инициализировать объекты Box.
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    double vol;

    // Получить объем первой коробки.
    vol = mybox1.volume();
    System.out.println("Объем равен " + vol);

    // Получить объем второй коробки.
    vol = mybox2.volume();
    System.out.println("Объем равен " + vol);
  }
}
```

После запуска программа генерирует следующий вывод:

```
Конструирование Box
Конструирование Box
Объем равен 1000.0
Объем равен 1000.0
```

Как видите, объекты `mybox1` и `mybox2` были инициализированы конструктором `Box()` при их создании. Поскольку конструктор дает всем блокам одинаковые размеры, $10 \times 10 \times 10$, объекты коробок, представленные с помощью `mybox1` и `mybox2`, будут иметь одинаковый объем. Оператор `println()` внутри `Box()` служит только в целях иллюстрации. Большинство конструкторов ничего не отображают. Они просто инициализируют объект.

Прежде чем двигаться дальше, давайте еще раз взглянем на операцию `new`. Вам уже известно, что при размещении объекта в памяти применяется показанная ниже общая форма:

```
переменная-класса = new имя-класса ();
```

Теперь вы понимаете, зачем нужны скобки после имени класса. На самом деле происходит вызов конструктора класса. Таким образом, в следующей строке:

```
Box mybox1 = new Box();
```

фрагмент `new Box()` вызывает конструктор `Box()`. Если конструктор для класса не определяется явно, тогда компилятор Java создает стандартный

конструктор. Вот почему предыдущая строка кода работала в более ранних версиях класса Box, в которых конструктор не определялся. При использовании стандартного конструктора все неинициализированные переменные экземпляра будут иметь стандартные значения, которые для числовых типов, ссылочных типов и логических значений равны соответственно нулю, null и false. Стандартного конструктора часто оказывается достаточно для простых классов, но для более сложных классов он обычно не подходит. После определения собственного конструктора стандартный конструктор больше не применяется.

Параметризованные конструкторы

Несмотря на то что конструктор Box() в предыдущем примере инициализирует объект Box, он не особенно полезен — все коробки имеют одинаковые размеры. Необходим способ создания объектов Box различных размеров. Простое решение заключается в добавлении параметров к конструктору. Как вы наверняка догадались, это сделает его гораздо более полезным. Например, в представленной далее версии класса Box определен параметризованный конструктор, который устанавливает размеры коробки в соответствии с указанными параметрами. Обратите особое внимание на то, как создаются объекты Box.

```
/* Здесь в Box используется параметризованный конструктор
   для инициализации размеров коробки.
*/

class Box {
  double width;
  double height;
  double depth;

  // Это конструктор для Box.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // Вычислить и возвратить объем.
  double volume() {
    return width * height * depth;
  }
}

class BoxDemo7 {
  public static void main(String[] args) {
    // Объявить, разместить в памяти и инициализировать объекты Box.
    mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box(3, 6, 9);

    double vol;

    // Получить объем первой коробки.
    vol = mybox1.volume();
    System.out.println("Объем равен " + vol);
```

```
    // Получить объем второй коробки.
    vol = mybox2.volume();
    System.out.println("Объем равен " + vol);
  }
}
```

Вот вывод, генерируемый программой:

```
Объем равен 3000.0
Объем равен 162.0
```

Как видите, каждый объект инициализируется в соответствии с тем, что указано в параметрах его конструктора. Например, в следующей строке:

```
Box mybox1 = new Box(10, 20, 15);
```

при создании операцией `new` объекта конструктору `Box()` передаются значения 10, 20 и 15. Таким образом, копии переменных экземпляра `width`, `height` и `depth` объекта `mybox1` будут содержать значения 10, 20 и 15.

Ключевое слово this

Иногда метод должен ссылаться на объект, на котором он вызывается. Для этого в Java определено ключевое слово `this`. Его можно использовать внутри любого метода для ссылки на *текущий* объект, т.е. `this` всегда будет ссылкой на объект, на котором был вызван метод. Ключевое слово `this` можно применять везде, где разрешена ссылка на объект типа текущего класса.

Чтобы лучше понять, на что ссылается `this`, взгляните на приведенную ниже версию конструктора `Box()`:

```
// Избыточное использование this.
Box(double w, double h, double d) {
  this.width = w;
  this.height = h;
  this.depth = d;
}
```

Данная версия `Box()` работает точно так же, как и предыдущая версия. Применение `this` избыточно, но совершенно корректно. Внутри `Box()` ключевое слово `this` всегда будет ссылаться на вызывающий объект. Хотя здесь `this` избыточно, оно полезно в других контекстах, один из которых объясняется в следующем разделе.

Сокрытие переменных экземпляра

Как вы знаете, в Java запрещено объявлять две локальные переменные с одинаковыми именами внутри той же самой или объемлющей области видимости. Интересно отметить, что у вас могут быть локальные переменные, в том числе формальные параметры методов, имена которых совпадают с именами переменных экземпляра класса. Однако когда локальная переменная имеет такое же имя, как у переменной экземпляра, то локальная пере-

менная *скрывает* переменную экземпляра. Вот почему width, height и depth не использовались в качестве имен параметров конструктора Box() внутри класса Box. Если бы они были выбраны, то имя width, например, ссылалось бы на формальный параметр, скрывая переменную экземпляра width. Хотя обычно проще применять отличающиеся имена, есть и другой выход из такой ситуации. Поскольку ключевое слово this позволяет ссылаться прямо на объект, его можно использовать для устранения любых конфликтов имен, которые могут возникать между переменными экземпляра и локальными переменными. Например, вот еще одна версия конструктора Box(), в которой имена width, height и depth применяются для параметров, а посредством this организуется доступ к переменным экземпляра с теми же именами:

```
// Использование this для устранения конфликтов имен.
Box(double width, double height, double depth) {
  this.width = width;
  this.height = height;
  this.depth = depth;
}
```

Есть одно предостережение: использование this в таком контексте иногда может сбивать с толку. Некоторые программисты стараются не применять для локальных переменных и формальные параметров такие имена, которые приводят к сокрытию переменных экземпляра. Разумеется, другие программисты уверены в обратном: они полагают, что использование тех же самых имен делает код яснее и применяют this для преодоления сокрытия переменных экземпляра. То, какой подход вы выберете — дело личного вкуса.

Сборка мусора

Поскольку объекты динамически размещаются в памяти с помощью операции new, вас может интересовать, каким образом такие объекты уничтожаются, и занимаемая ими память освобождается с целью последующего выделения. В языках, подобных традиционному C++, динамически размещенные объекты необходимо освобождать вручную с помощью операции delete. В Java используется другой подход; освобождение поддерживается автоматически. Методика, которая позволяет это делать, называется *сборкой мусора*. Она работает следующим образом: когда ссылок на объект не существует, то считается, что такой объект больше не нужен, и занимаемая им память может быть освобождена. Нет необходимости явно уничтожать объекты. Сборка мусора происходит нерегулярно (если вообще происходит) во время выполнения программы. Она не инициируется просто потому, что существует один или несколько объектов, которые больше не используются. Кроме того, в разных реализациях исполняющей среды Java будут применяться варьирующиеся подходы к сборке мусора, но по большей части вам не придется думать о ней при написании своих программ.

Класс Stack

Хотя класс Box полезен для иллюстрации основных элементов класса, практической ценности от него мало. Чтобы продемонстрировать реальную мощь классов, в завершение главы будет рассмотрен более сложный пример. Как вы помните из обсуждения ООП, представленного в главе 2, одним из самых важных преимуществ ООП является инкапсуляция данных и кода, который манипулирует этими данными. Вы видели, что класс представляет собой механизм, с помощью которого достигается инкапсуляция в Java. Создавая класс, вы создаете новый тип данных, который определяет как природу обрабатываемых данных, так и процедуры, используемые для их обработки. Кроме того, методы определяют согласованный и контролируемый интерфейс к данным класса. Таким образом, вы можете работать с классом через его методы, не беспокоясь о деталях его реализации или о том, каким образом происходит фактическое управление данными внутри класса. В некотором смысле класс похож на "механизм обработки данных". Для управления таким механизмом не требуется никаких знаний о том, что происходит внутри него. На самом деле, поскольку детали скрыты, внутреннюю работу механизма можно по мере необходимости изменять. До тех пор, пока класс эксплуатируется в коде через его методы, внутренние детали могут изменяться, не вызывая побочных эффектов за пределами класса.

Чтобы взглянуть на практическое воплощение предыдущего обсуждения, давайте разработаем один из типовых примеров инкапсуляции: стек. *Стек* сохраняет данные по принципу "последний пришел — первым обслужен", т.е. стек подобен стопке тарелок на столе: первая тарелка, поставленная на стол, используется последней. Стеки управляются с помощью двух операций, традиционно называемых *помещением* и *извлечением*. Чтобы сохранить элемент на верхушке стека, будет применяться помещение. Чтобы взять элемент из стека, будет использоваться извлечение. Как вы увидите, инкапсулировать полный механизм стека несложно.

Ниже показан код класса по имени Stack, который реализует стек вместительностью до десяти целых чисел:

```
// Этот класс реализует стек целых чисел, который может хранить 10 значений
class Stack {
  int[] stck = new int[10];
  int tos;
  // Инициализировать верхушку стека.
  Stack() {
    tos = -1;
  }
  // Поместить элемент в стек.
  void push(int item) {
    if(tos==9)
      System.out.println("Стек полон.");
    else
      stck[++tos] = item;
  }
```

```
  // Извлечь элемент из стека.
  int pop() {
    if(tos < 0) {
      System.out.println("Стек опустошен.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

Здесь видно, что в классе `Stack` определены два элемента данных, два метода и конструктор. Стек целых чисел хранится в массиве `stck`. Этот массив индексируется переменной `tos`, которая всегда содержит индекс верхушки стека. Конструктор `Stack()` инициализирует `tos` значением `-1`, что указывает на пустой стек. Метод `push()` помещает элемент в стек. Для получения элемента предназначен метод `pop()`. Так как доступ к стеку осуществляется через методы `push()` и `pop()`, тот факт, что стек хранится в массиве, на самом деле не имеет отношения к использованию стека. Например, стек может храниться в более сложной структуре данных, такой как связный список, но интерфейс, определяемый методами `push()` и `pop()`, останется прежним.

Приведенный далее класс `TestStack` демонстрирует работу с классом `Stack`. В нем создаются два стека целых чисел, после чего в каждый помещается несколько значений, которые затем извлекаются.

```
class TestStack {
  public static void main(String[] args) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

    // Поместить несколько чисел в стеки.
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);

    // Извлечь эти числа из стеков.
    System.out.println("Стек в mystack1:");
    for(int i=0; i<10; i++)
      System.out.println(mystack1.pop());

    System.out.println("Стек в mystack2:");
    for(int i=0; i<10; i++)
      System.out.println(mystack2.pop());
  }
}
```

Программа генерирует такой вывод:

```
Стек в mystack1:
9
8
7
6
5
4
```

```
3
2
1
0
Стек в mystack2:
19
18
17
16
15
14
13
12
11
10
```

Легко заметить, что содержимое каждого стека является обособленным.

И последнее замечание, касающееся класса `Stack`. При его текущей реализации массив, хранящий стек (`stck`), может быть изменен кодом вне класса `Stack`. В итоге существует риск неправильного использования или повреждения класса `Stack`. В следующей главе вы увидите, как устранить проблему.

ГЛАВА

7 Подробный анализ методов и классов

В этой главе продолжается обсуждение методов и классов, которое началось в предыдущей главе. В ней рассматривается несколько связанных с методами тем, включая перегрузку, передачу параметров и рекурсию. Затем мы снова обратимся к классам и обсудим управление доступом, использование ключевого слова `static` и работу с одним из самых важных встроенных классов Java: `String`.

Перегрузка методов

Язык Java разрешает определять в одном классе два и более метода, которые имеют одно и то же имя, если их объявления параметров отличаются. В таком случае говорят, что методы перегружены, а сам процесс называется *перегрузкой методов*. Перегрузка методов — один из способов поддержки полиморфизма в Java. Если вы никогда не имели дело с языком, допускающим перегрузку методов, то поначалу эта концепция может показаться странной. Но, как вы увидите, перегрузка методов относится к самым захватывающим и полезным возможностям Java.

При вызове перегруженного метода компилятор Java использует тип и/или количество аргументов в качестве ориентира, чтобы определить, какую версию перегруженного метода фактически вызывать. Таким образом, перегруженные методы должны отличаться типом и/или количеством параметров. Хотя перегруженные методы могут возвращать разные типы, одного типа возвращаемого значения недостаточно, чтобы различить две версии метода. Когда компилятор Java встречает вызов перегруженного метода, он просто выполняет версию метода, параметры которой соответствуют аргументам, указанным в вызове.

Ниже приведен простой пример, иллюстрирующий перегрузку методов:

```
// Демонстрация перегрузки методов.
class OverloadDemo {
  void test() {
    System.out.println("Параметры отсутствуют");
  }
```

```
  // Перегрузить test() для одного целочисленного параметра.
  void test(int a) {
    System.out.println("a: " + a);
  }

  // Перегрузить test() для двух целочисленных параметров.
  void test(int a, int b) {
    System.out.println("a и b: " + a + " " + b);
  }

  // Перегрузить test() для одного параметра типа double.
  double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}

class Overload {
  public static void main(String[] args) {
    OverloadDemo ob = new OverloadDemo();
    double result;

    // Вызвать все версии test().
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.25);
    System.out.println("Результат вызова ob.test(123.25): " + result);
  }
}
```

Программа сгенерирует следующий вывод:

```
Параметры отсутствуют
a: 10
a и b: 10 20
double a: 123.25
Результат вызова ob.test(123.25): 15190.5625
```

Как видите, метод `test()` перегружается четыре раза. Первая версия не принимает параметров, вторая принимает один целочисленный параметр, третья — два целочисленных параметра, а четвертая — один параметр `double`. Тот факт, что четвертая версия `test()` также возвращает значение, не имеет значения для перегрузки, поскольку возвращаемые типы не играют роли в распознавании перегруженных методов.

При вызове перегруженного метода компилятор Java ищет соответствие между аргументами, используемыми для вызова метода, и параметрами метода. Однако это совпадение не всегда должно быть точным. В некоторых случаях автоматическое преобразование типов в Java может играть роль в распознавании перегруженных методов. Например, взгляните на следующую программу:

```
// При перегрузке применяется автоматическое преобразование типов.
class OverloadDemo {
  void test() {
    System.out.println("Параметры отсутствуют");
  }
```

```
  // Перегрузить test() для двух целочисленных параметров.
  void test(int a, int b) {
    System.out.println("a и b: " + a + " " + b);
  }

  // Перегрузить test() для одного параметра типа double.
  void test(double a) {
    System.out.println("Внутри test(double) a: " + a);
  }
}

class Overload {
  public static void main(String[] args) {
    OverloadDemo ob = new OverloadDemo();
    int i = 88;

    ob.test();
    ob.test(10, 20);

    ob.test(i);          // будет вызываться test(double)
    ob.test(123.2);      // будет вызываться test(double)
  }
}
```

Вот вывод, генерируемый программой:

```
Параметры отсутствуют
a и b: 10 20
Внутри test(double) a: 88.0
Внутри test(double) a: 123.2
```

Легко заметить, что в данной версии `OverloadDemo` метод `test(int)` не определен. Тогда при вызове `test()` с целочисленным аргументом внутри `Overload` подходящая версия метода не будет найдена. Тем не менее, компилятор Java способен автоматически преобразовывать целое число в число типа `double`, и такое преобразование можно использовать для распознавания вызова. Поэтому после того, как `test(int)` не найден, компилятор Java повышает `i` до `double` и затем вызывает `test(double)`. Конечно, если бы версия `test(int)` была определена, то именно она была бы вызвана. Компилятор Java задействует свои автоматические преобразования типов только при отсутствии точного совпадения.

Перегрузка методов поддерживает полиморфизм, т.к. он представляет собой один из способов, которым в Java реализуется парадигма "один интерфейс, несколько методов". Давайте выясним, каким образом. В языках, не поддерживающих перегрузку методов, каждому методу должно быть назначено уникальное имя. Однако часто вам потребуется реализовать по существу один и тот же метод для разных типов данных. Рассмотрим функцию для абсолютного значения. В языках, не поддерживающих перегрузку, обычно существует три или более версий такой функции, каждая из которых имеет слегка отличающееся имя. Например, в языке C функция `abs()` возвращает абсолютное значение целого числа, `labs()` — абсолютное значение длинного целого числа, а `fabs()` — абсолютное значение значения с плавающей точкой. Поскольку перегрузка в C не поддерживается, каждая функция имеет

собственное имя, хотя все три функции выполняют, по сути, одну и ту же работу, что делает ситуацию концептуально более сложной, чем она есть на самом деле. Хотя базовая концепция каждой функции одна и та же, вам нужно запомнить все три имени. В языке Java ситуация подобного рода не возникает, потому что каждый метод для абсолютного значения может иметь одно и то же имя. Действительно, стандартная библиотека классов Java включает метод получения абсолютного значения, называемый `abs()`. Этот метод перегружен в классе `Math` для обработки всех числовых типов. Компилятор Java определяет, какую версию `abs()` вызывать, основываясь на типе аргумента.

Ценность перегрузки обусловлена тем, что она позволяет получить доступ к связанным методам с применением общего имени. Таким образом, имя `abs` представляет выполняемое *общее действие*. Выбор правильной *конкретной* версии в сложившихся обстоятельствах возлагается на компилятор. Вам как программисту достаточно лишь запомнить общую выполняемую операцию. Благодаря полиморфизму несколько имен были сведены в одно. Хотя приведенный пример довольно прост, если вы расширите концепцию, то увидите, каким образом перегрузка может помочь справиться с более высокой сложностью.

В случае перегрузки метода каждая его версия может выполнять любые желаемые действия. Нет правила, утверждающего о том, что перегруженные методы должны быть связаны друг с другом. Тем не менее, со стилистической точки зрения перегрузка методов подразумевает наличие взаимоотношения между ними. Таким образом, хотя и допускается использовать одно и то же имя для перегрузки несвязанных методов, вы не должны поступать так. Скажем, вы можете выбрать имя `sqr` при создании методов, возвращающих *квадрат* целого числа и *квадратный корень* значения с плавающей запятой. Но эти две операции принципиально разные. Применение перегрузки метода в подобной манере противоречит его первоначальной цели. На практике следует перегружать только тесно связанные операции.

Перегрузка конструкторов

Помимо перегрузки обычных методов вы также можете перегружать методы конструкторов. Фактически для большинства создаваемых вами реальных классов перегруженные конструкторы будут нормой, а не исключением. Чтобы выяснить причину, вернемся к классу `Box`, разработанному в предыдущей главе. Ниже представлена последняя версия `Box`:

```
class Box {
  double width;
  double height;
  double depth;

  // Это конструктор для Box.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
```

```
  // Вычислить и возвратить объем.
  double volume() {
    return width * height * depth;
  }
}
```

Как видите, конструктор `Box()` требует трех параметров, т.е. все объявления объектов `Box` должны передавать конструктору `Box()` три аргумента. Например, следующий оператор в настоящее время недопустим:

```
Box ob = new Box();
```

Поскольку вызов `Box()` требует передачи трех аргументов, вызов без них приведет к ошибке. В результате возникает ряд важных вопросов. Что, если вам просто необходим объект коробки, первоначальные размеры которой не важны (или не известны)? Или что, если вы хотите иметь возможность инициализировать объект кубика, указав только одно значение, которое будет использоваться для всех трех измерений? В том виде, в каком сейчас записан класс `Box`, такие варианты совершенно не доступны.

К счастью, решить упомянутые проблемы довольно просто: нужно лишь перегрузить конструктор класса `Box`, чтобы он обрабатывал описанные выше ситуации. Ниже приведена программа, содержащая усовершенствованную версию `Box`, которая именно это и делает:

```
/* Здесь в классе Box определены три конструктора для инициализации
   размеров объекта коробки различными способами.
*/
class Box {
  double width;
  double height;
  double depth;

  // Конструктор, используемый в случае указания всех размеров.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // Конструктор, применяемый в случае, если размеры вообще не указаны.
  Box() {
    width = -1;      // использовать -1 для обозначения
    height = -1;     // неинициализированного
    depth = -1;      // объекта коробки
  }

  // Конструктор, используемый в случае создания объекта кубика.
  Box(double len) {
    width = height = depth = len;
  }

  // Вычислить и возвратить объем.
  double volume() {
    return width * height * depth;
  }
}
```

```
class OverloadCons {
  public static void main(String[] args) {
    // Создать объекты коробок с применением различных конструкторов.
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);
    double vol;

    // Вычислить объем первой коробки.
    vol = mybox1.volume();
    System.out.println("Объем mybox1 равен " + vol);

    // Вычислить объем второй коробки.
    vol = mybox2.volume();
    System.out.println("Объем mybox2 равен " + vol);

    // Вычислить объем кубика.
    vol = mycube.volume();
    System.out.println("Объем mycube равен " + vol);
  }
}
```

Вот вывод, генерируемый программой:

```
Объем mybox1 равен 3000.0
Объем mybox2 равен -1.0
Объем mycube равен 343.0
```

Как видите, на основе аргументов, указанных при выполнении new, вызывается надлежащий перегруженный конструктор.

Использование объектов в качестве параметров

До сих пор в качестве параметров методов применялись только простые типы. Однако передача объектов методам является правильной и распространенной практикой. Например, возьмем показанную далее короткую программу:

```
// Объекты можно передавать методам.
class Test {
  int a, b;
  Test(int i, int j) {
    a = i;
    b = j;
  }
  // Возвратить true, если объект o равен вызывающему объекту.
  boolean equalTo(Test o) {
    if(o.a == a && o.b == b) return true;
    else return false;
  }
}
class PassOb {
  public static void main(String[] args) {
    Test ob1 = new Test(100, 22);
    Test ob2 = new Test(100, 22);
    Test ob3 = new Test(-1, -1);
```

```
    System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
    System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
  }
}
```

Программа генерирует следующий вывод:

```
ob1 == ob2: true
ob1 == ob3: false
```

Как видите, метод `equalTo()` внутри `Test` сравнивает два объекта на предмет равенства и возвращает результат, т.е. он сравнивает вызывающий объект с тем, который ему передается. Если они содержат одинаковые значения, тогда метод возвращает `true`. В противном случае возвращается `false`. Обратите внимание, что параметр `o` в `equalTo()` указывает `Test` в качестве своего типа. Хотя `Test` — тип класса, созданный программой, он используется точно так же, как и встроенные типы Java.

Одно из наиболее распространенных применений параметров объекта связано с конструкторами. Часто требуется создать новый объект так, чтобы он изначально был таким же, как какой-то существующий объект. Для этого необходимо определить конструктор, который принимает объект своего класса в качестве параметра. Например, следующая версия `Box` позволяет одному объекту инициализировать другой:

```
// Здесь класс Box позволяет один объект инициализировать другим.
class Box {
  double width;
  double height;
  double depth;

  //Обратите внимание на этот конструктор, который принимает объект типа Box
  Box(Box ob) {  // передать объект конструктору
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // Конструктор, используемый в случае указания всех размеров.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // Конструктор, применяемый в случае, если размеры вообще не указаны.
  Box() {
    width = -1;      // использовать -1 для обозначения
    height = -1;     // неинициализированного
    depth = -1;      // объекта коробки
  }

  // Конструктор, используемый в случае создания объекта кубика.
  Box(double len) {
    width = height = depth = len;
  }
```

```
  // Вычислить и возвратить объем.
  double volume() {
    return width * height * depth;
  }
}
class OverloadCons2 {
  public static void main(String[] args) {
    // Создать объекты коробок с применением различных конструкторов.
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);
    Box myclone = new Box(mybox1);  // создать копию объекта mybox1
    double vol;
    // Вычислить объем первой коробки.
    vol = mybox1.volume();
    System.out.println("Объем mybox1 равен " + vol);

    // Вычислить объем второй коробки.
    vol = mybox2.volume();
    System.out.println("Объем mybox2 равен " + vol);

    // Вычислить объем кубика.
    vol = mycube.volume();
    System.out.println("Объем mycube равен " + vol);

    // Вычислить объем копии.
    vol = myclone.volume();
    System.out.println("Объем копии равен " + vol);
  }
}
```

Начав создавать собственные классы, вы увидите, что для удобного и эффективного создания объектов обычно требуется множество форм конструкторов.

Подробный анализ передачи аргументов

Говоря в общем, в языках программирования существуют два способа, которыми можно передавать аргумент подпрограмме. Первый способ — *вызов по значению*, при котором в формальный параметр подпрограммы копируется *значение* аргумента, поэтому изменения, вносимые в параметр подпрограммы, не влияют на аргумент. Второй способ — *вызов по ссылке*. При таком подходе в параметр передается *ссылка* на аргумент (а не его значение). Внутри подпрограммы эта ссылка используется для доступа к фактическому аргументу, указанному в вызове, т.е. изменения, вносимые в параметр, повлияют на аргумент, который применялся при вызове подпрограммы. Вы увидите, что хотя в Java для передачи всех аргументов используется вызов по значению, точный результат зависит от того, какой тип передается — примитивный или ссылочный.

Когда методу передается примитивный тип, то происходит передача по значению. Таким образом, создается копия аргумента, и все то, что делается с параметром, получающим аргумент, не имеет никакого эффекта вне метода.

Например, рассмотрим следующую программу:

```
// Примитивные типы передаются по значению.
class Test {
  void meth(int i, int j) {
    i *= 2;
    j /= 2;
  }
}

class CallByValue {
  public static void main(String[] args) {
    Test ob = new Test();

    int a = 15, b = 20;

    System.out.println("a и b перед вызовом: " +
                       a + " " + b);

    ob.meth(a, b);

    System.out.println("a и b после вызова: " +
                       a + " " + b);
  }
}
```

Вот вывод из программы:

```
a и b перед вызовом: 15 20
a и b после вызова: 15 20
```

Как видите, операции, выполняемые внутри `meth()`, не влияют на значения `a` и `b`, используемые в вызове; их значения здесь не изменились на `30` и `10`.

Когда методу передается объект, ситуация кардинально меняется, потому что объекты передаются посредством того, что фактически называется вызовом по ссылке. Имейте в виду, что при создании переменной типа класса создается только ссылка на объект. Таким образом, когда такая ссылка передается методу, то параметр, который ее получает, будет ссылаться на тот же объект, на который ссылается аргумент. Фактически это означает, что объекты действуют так, будто они передаются методам с помощью вызова по ссылке. Изменения объекта внутри метода *влияют* на объект, указанный в качестве аргумента. Например, взгляните на показанную ниже программу:

```
// Объекты передаются через ссылки на них.
class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }

  // Передать объект.
  void meth(Test o) {
    o.a *= 2;
    o.b /= 2;
  }
}
```

```
class PassObjRef {
  public static void main(String[] args) {
    Test ob = new Test(15, 20);

    System.out.println("ob.a и ob.b перед вызовом: " +
                       ob.a + " " + ob.b);
    ob.meth(ob);

    System.out.println("ob.a и ob.b после вызова: " + ob.a + " " + ob.b);
  }
}
```

Программа генерирует следующий вывод:

```
ob.a и ob.b перед вызовом: 15 20
ob.a и ob.b после вызова: 30 10
```

Несложно заметить, что в данном случае действия внутри `meth()` воздействуют на объект, используемый в качестве аргумента.

Помните! При передаче методу ссылки на объект сама ссылка передается с применением вызова по значению. Но поскольку передаваемое значение относится к объекту, копия этого значения по-прежнему будет ссылаться на тот же объект, что и соответствующий аргумент.

Возвращение объектов

Метод способен возвращать данные любого типа, включая типы классов, которые вы создаете. Например, в приведенной далее программе метод `incrByTen()` возвращает объект, в котором значение `a` на 10 больше, чем в вызывающем объекте.

```
// Возвращение объекта.
class Test {
  int a;

  Test(int i) {
    a = i;
  }

  Test incrByTen() {
    Test temp = new Test(a+10);
    return temp;
  }
}
class RetOb {
  public static void main(String[] args) {
    Test ob1 = new Test(2);
    Test ob2;

    ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);

    ob2 = ob2.incrByTen();
    System.out.println("ob2.a после второго увеличения: " + ob2.a);
  }
}
```

Вот вывод, генерируемый программой:

```
ob1.a: 2
ob2.a: 12
ob2.a после второго увеличения: 22
```

Как видите, каждый раз, когда вызывается `incrByTen()`, создается новый объект, а вызывающей процедуре возвращается ссылка на него.

В предыдущей программе продемонстрирован еще один важный момент: поскольку все объекты динамически размещаются с помощью операции `new`, вам не нужно беспокоиться о том, что объект выйдет за пределы области видимости, т.к. метод, в котором он был создан, завершается. Объект будет продолжать существовать до тех пор, пока в программе где-то есть ссылка на него. Когда ссылки на него исчезнут, занимаемая объектом память освободится при очередной сборке мусора.

Рекурсия

В языке Java поддерживается *рекурсия* — процесс определения чего-либо в терминах самого себя. Что касается программирования на Java, то рекурсия является характерной чертой, позволяющей методу вызывать самого себя. Метод, который вызывает сам себя, называется *рекурсивным*.

Классическим примером рекурсии считается вычисление факториала числа. Факториал числа N — это произведение всех целых чисел от 1 до N. Например, факториал 3 равен $1 \times 2 \times 3$, или 6. Рассмотрим, как можно вычислить факториал с помощью рекурсивного метода:

```
// Простой пример использования рекурсии.
class Factorial {
  // Рекурсивный метод.
  int fact(int n) {
    int result;

    if(n==1) return 1;
    result = fact(n-1) * n;
    return result;
  }
}

class Recursion {
  public static void main(String[] args) {
    Factorial f = new Factorial();

    System.out.println("Факториал 3 равен " + f.fact(3));
    System.out.println("Факториал 4 равен " + f.fact(4));
    System.out.println("Факториал 5 равен " + f.fact(5));
  }
}
```

Ниже показан вывод, генерируемый программой:

```
Факториал 3 равен 6
Факториал 4 равен 24
Факториал 5 равен 120
```

Если вы не знакомы с рекурсивными методами, то работа `fact()` может показаться немного запутанной. Давайте проясним детали. Когда `fact()` вызывается с аргументом 1, функция возвращает 1; в противном случае возвращается произведение `fact(n-1)*n`. Для вычисления такого выражения вызывается `fact()` с параметром `n-1`. Процесс повторяется до тех пор, пока значение `n` не станет равным 1 и не начнется возврат из вызовов метода.

Чтобы лучше понять, как функционирует метод `fact()`, рассмотрим небольшой пример. При вычислении факториала 3 первый вызов `fact()` приводит ко второму вызову с аргументом 2, а тот в свою очередь — к третьему вызову `fact()` с аргументом 1. Третий вызов возвратит значение 1, которое затем умножается на 2 (значение `n` во втором вызове). Результат (равный 2) затем возвращается в исходный вызов `fact()` и умножается на 3 (первоначальное значение `n`), что дает ответ 6. Возможно, вам будет интересно вставить операторы `println()` в `fact()`, которые будут показывать, на каком уровне находится каждый вызов и каковы промежуточные ответы.

Когда метод вызывает сам себя, новые локальные переменные и параметры размещаются в стеке, а код метода выполняется с этими новыми переменными с самого начала. При возврате из каждого рекурсивного вызова старые локальные переменные и параметры удаляются из стека, и выполнение возобновляется в точке вызова внутри метода. Можно сказать, что рекурсивные методы работают в стиле "раздвижения" и "складывания" подзорной трубы.

Рекурсивные версии многих подпрограмм могут выполняться немного медленнее своих итеративных эквивалентов из-за добавочных накладных расходов на дополнительные вызовы методов. Большое количество рекурсивных вызовов метода может привести к переполнению стека. Поскольку хранилище для параметров и локальных переменных находится в стеке, и каждый новый вызов создает новую копию этих переменных, вполне возможно, что стек исчерпается. В таком случае исполняющая среда Java инициирует исключение. Тем не менее, как правило, проблема не возникает, если рекурсивная подпрограмма не выходит из-под контроля.

Главное преимущество рекурсивных методов связано с тем, что их можно использовать для создания более ясных и простых версий ряда алгоритмов, чем их итеративные аналоги. Скажем, алгоритм быстрой сортировки довольно сложно реализовать итеративным способом. Кроме того, некоторые типы алгоритмов из области искусственного интеллекта проще всего реализовать с помощью рекурсивных решений.

При написании рекурсивных методов вы должны где-то предусмотреть оператор `if`, чтобы заставить метод выполнить возврат без рекурсивного вызова. Если вы этого не сделаете, то после вызова возврат из метода никогда не произойдет. Такая ошибка весьма распространена, когда приходится иметь дело с рекурсией. Свободно применяйте операторы `println()` во время разработки, чтобы вы могли следить за происходящим и прерывать выполнение, если видите, что допустили ошибку.

Вот еще один пример рекурсии. Рекурсивный метод `printArray()` выводит первые `i` элементов в массиве `values`.

```
// Еще один пример использования рекурсии.
class RecTest {
  int[] values;

  RecTest(int i) {
    values = new int[i];
  }

  // Рекурсивно отобразить элементы массива.
  void printArray(int i) {
    if(i==0) return;
    else printArray(i-1);
    System.out.println("[" + (i-1) + "] " + values[i-1]);
  }
}

class Recursion2 {
  public static void main(String[] args) {
    RecTest ob = new RecTest(10);
    int i;

    for(i=0; i<10; i++) ob.values[i] = i;

    ob.printArray(10);
  }
}
```

Программа генерирует следующий вывод:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

Введение в управление доступом

Как вам уже известно, инкапсуляция связывает данные с кодом, который ими манипулирует. Однако инкапсуляция предоставляет еще одно важное средство: *управление доступом*. С помощью инкапсуляции вы можете контролировать то, какие части программы могут обращаться к членам класса. Управляя доступом, можно предотвратить неправильную эксплуатацию. Например, за счет разрешения доступа к данным только через четко определенный набор методов вы можете предотвратить неправомерное использование этих данных. Таким образом, при правильной реализации класс создает "черный ящик", с которым можно взаимодействовать, но нарушить его внутреннюю работу не удастся. Тем не менее, представленные ранее классы не полностью отвечают

такой цели. Например, возьмем класс `Stack`, показанный в конце главы 6. Хотя методы `push()` и `pop()` действительно обеспечивают управляемый интерфейс для стека, данный интерфейс не применяется принудительно, т.е. другая часть программы может обойти указанные методы и обратиться к стеку напрямую. Разумеется, в неумелых руках это может привести к неприятностям. В настоящем разделе вы ознакомитесь с механизмом, с помощью которого сможете точно управлять доступом к различным членам класса.

Доступ к члену определяется *модификатором доступа*, присоединенным к его объявлению. Язык Java предлагает богатый набор модификаторов доступа. Некоторые аспекты управления доступом в основном связаны с наследованием или пакетами. (Пакет по существу представляет собой группу классов.) Эти части механизма управления доступом в Java будут обсуждаться в последующих главах. Давайте начнем с изучения управления доступом применительно к одному классу. Как только вы поймете основы управления доступом, остальное дастся легко.

На заметку! Средство модулей, добавленное в JDK 9, тоже может влиять на доступность. Модули будут описаны в главе 16.

Модификаторами доступа Java являются `public` (открытый), `private` (закрытый) и `protected` (защищенный). В Java также определен стандартный уровень доступа. Модификатор доступа `protected` применяется, только когда задействовано наследование. Другие модификаторы доступа описаны далее.

Давайте начнем с определения `public` и `private`. Когда член класса изменяется с помощью `public`, доступ к нему может получать любой другой код. Когда член класса указан как `private`, доступ к нему могут получать только другие члены этого класса. Теперь вы понимаете, почему объявлению метода `main()` всегда предшествовал модификатор `public`. Он вызывается кодом, находящимся вне программы, т.е. исполняющей средой Java. Если модификатор доступа не задействован, то по умолчанию член класса является открытым в своем пакете, но к нему нельзя получить доступ за пределами пакета. (Пакеты обсуждаются в главе 9.)

В разработанных до сих пор классах все члены класса использовали стандартный режим доступа. Однако это не то, что вы обычно хотели бы иметь. Как правило, вам нужно ограничить доступ к членам данных класса, разрешив доступ только через методы. Кроме того, будут случаи, когда вы пожелаете определить методы, которые являются закрытыми для класса.

Модификатор доступа предшествует остальной части спецификации типа члена. Другими словами, с него должен начинаться оператор объявления члена. Вот пример:

```
public int i;
private double j;

private int myMethod(int a, char b) {  //...
```

Чтобы понять влияние открытого и закрытого доступа, рассмотрим следующую программу:

```
/* В этой программе демонстрируется
   отличие между public и private.
*/
class Test {
  int a;                // стандартный доступ
  public int b;         // открытый доступ
  private int c;        // закрытый доступ

  // методы для доступа к c
  void setc(int i) {    // установить значение c
    c = i;
  }
  int getc() {          // получить значение c
    return c;
  }
}

class AccessTest {
  public static void main(String[] args) {
    Test ob = new Test();

    // Поступать так законно, т.к. к членам a и b разрешен прямой доступ.
    ob.a = 10;
    ob.b = 20;

    // Поступать так нельзя, т.к. возникнет ошибка.
    // ob.c = 100;      // Ошибка!

    // Получать доступ к члену c необходимо через его методы.
    ob.setc(100);       // нормально
    System.out.println("a, b и c: " + ob.a + " " +
                       ob.b + " " + ob.getc());
  }
}
```

Как видите, внутри класса `Test` применяется стандартный доступ, что в данном примере равнозначно указанию `public`. Член `b` явно определен как `public`, а член `c` — как `private`. Это означает, что доступ к члену `c` в коде за пределами его класса невозможен. Таким образом, внутри класса `AccessTest` член `c` нельзя использовать напрямую. Доступ к нему должен осуществляться через его открытые методы: `setc()` и `getc()`. Если бы вы удалили символ комментария в начале следующей строки:

```
// ob.c = 100;  // Ошибка!
```

то не смогли бы скомпилировать программу из-за нарушения прав доступа.

Чтобы увидеть, как применить управление доступом к более реальному примеру, рассмотрим показанную в конце главы 6 усовершенствованную версию класса `Stack`:

```
// Этот класс реализует стек целых чисел, который может хранить 10 значений
class Stack {
  /* Теперь stck и tos являются закрытыми. Это значит,
     что они не могут быть случайно или злонамеренно
     изменены таким образом, что может повредиться стек.
  */
```

```
  private int[] stck = new int[10];
  private int tos;
  // Инициализировать верхушку стека.
  Stack() {
    tos = -1;
  }
  // Поместить элемент в стек.
  void push(int item) {
    if(tos==9)
      System.out.println("Стек полон.");
    else
      stck[++tos] = item;
  }
  // Извлечь элемент из стека.
  int pop() {
    if(tos < 0) {
      System.out.println("Стек опустошен.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

Здесь видно, что и член `stck`, который хранит стек, и член `tos`, представляющий индекс верхушки стека, определены как `private`, т.е. к ним нельзя получить доступ либо изменить их, кроме как с помощью `push()` и `pop()`. Превращение `tos`, например, в закрытый член приводит к тому, что другие части программы не смогут непреднамеренно установить для него значение, выходящее за пределы массива `stck`.

В следующей программе демонстрируется использование усовершенствованного класса `Stack`. Попробуйте удалить закомментированные строки, чтобы удостовериться в том, что члены `stck` и `tos` действительно недоступны.

```
class TestStack {
  public static void main(String[] args) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();
    // Поместить несколько чисел в стеки.
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);
    // Извлечь эти числа из стеков.
    System.out.println("Стек в mystack1:");
    for(int i=0; i<10; i++)
      System.out.println(mystack1.pop());
    System.out.println("Стек в mystack2:");
    for(int i=0; i<10; i++)
      System.out.println(mystack2.pop());
    // Приведенные далее операторы являются недопустимыми.
    // mystack1.tos = -2;
    // mystack2.stck[3] = 100;
  }
}
```

Хотя методы обычно обеспечивают доступ к данным, которые определены в классе, так бывает не всегда. Совершенно правильно позволить переменной экземпляра быть открытой, когда на то имеется веская причина. Скажем, большинство несложных классов в этой книге ради простоты были созданы без особого беспокойства об управлении доступом к переменным экземпляра. Тем не менее, в большинстве реальных классов необходимо разрешать выполнение операций с данными только через методы. В следующей главе мы вернемся к теме управления доступом. Вы узнаете, что оно крайне важно при реализации наследования.

Ключевое слово `static`

Временами вам понадобится определять член класса, который будет применяться независимо от любого объекта данного класса. Обычно доступ к члену класса должен осуществляться только в сочетании с объектом его класса. Однако можно создать член, который можно использовать сам по себе, без привязки к конкретному экземпляру. Чтобы создать такой элемент, перед его объявлением следует указать ключевое слово `static` (статический). Когда член объявляется статическим, к нему можно получать доступ до того, как будут созданы какие-либо объекты его класса, и без ссылки на какой-либо объект. Объявить статическими можно как методы, так и переменные. Наиболее распространенным примером статического члена является метод `main()`, который объявлен как `static`, потому что он должен быть вызван до того, как будут созданы любые объекты.

Переменные экземпляра, объявленные как `static`, по существу являются глобальными переменными. При объявлении объектов такого класса копия статической переменной не создается. Взамен все экземпляры класса имеют дело с одной и той же статической переменной.

С методами, объявленными как `static`, связано несколько ограничений.

- Они могут напрямую вызывать только другие статические методы своего класса.
- Они могут напрямую получать доступ только к статическим переменным своего класса.
- Они никоим образом не могут ссылаться на `this` или `super`. (Ключевое слово `super` относится к наследованию и описано в следующей главе.)

Если для инициализации статических переменных нужно выполнять вычисления, тогда можно объявить блок `static`, который выполняется в точности один раз, когда класс загружается впервые. В примере ниже показан класс со статическим методом, несколькими статическими переменными и статическим блоком инициализации:

```
// Демонстрация применения статических переменных, методов и блоков.
class UseStatic {
  static int a = 3;
  static int b;
```

```
  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }
  static {
    System.out.println("Инициализация в статическом блоке.");
    b = a * 4;
  }
  public static void main(String[] args) {
    meth(42);
  }
}
```

Сразу после загрузки класса `UseStatic` запускаются все статические операторы. Сначала `a` устанавливается в 3, затем выполняется блок `static`, который выводит сообщение, после чего `b` инициализируется значением `a*4`, или 12. Далее вызывается метод `main()`, который вызывает `meth()`, передавая в `x` значение 42. Три оператора `println()` относятся к двум статическим переменным `a` и `b`, а также к параметру `x`.

Вот вывод, генерируемый программой:

```
Инициализация в статическом блоке.
x = 42
a = 3
b = 12
```

За пределами класса, в котором они определены, статические методы и переменные могут использоваться независимо от любого объекта. Для этого понадобится только указать имя их класса и за ним операцию точки. Скажем, если вы хотите вызвать статический метод вне его класса, то можете применить следующую общую форму:

```
имя-класса.метод()
```

Здесь в имя-класса указывается имя класса, где объявлен статический метод. Как видите, формат аналогичен тому, который используется для вызова нестатических методов через переменные ссылок на объекты. Доступ к статической переменной можно получить тем же способом — с помощью операции точки после имени класса. Именно так в Java реализована управляемая версия глобальных методов и глобальных переменных.

Рассмотрим пример. Внутри `main()` доступ к статическому методу `callme()` и статической переменной `b` осуществляется через имя их класса `StaticDemo`:

```
class StaticDemo {
  static int a = 42;
  static int b = 99;
  static void callme() {
    System.out.println("a = " + a);
  }
}
```

```
class StaticByName {
  public static void main(String[] args) {
    StaticDemo.callme();
    System.out.println("b = " + StaticDemo.b);
  }
}
```

Вывод программы выглядит так:

```
a = 42
b = 99
```

Ключевое слово `final`

Поле может быть объявлено как `final` (финальное), что предотвращает изменение его содержимого, делая его по существу константой. Это означает, что поле `final` должно быть инициализировано при его объявлении. Существуют два способа инициализации такого поля. Во-первых, полю `final` можно присвоить значение при его объявлении. Во-вторых, полю `final` можно присвоить значение в конструкторе. Первый подход, пожалуй, встречается наиболее часто. Вот пример:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Теперь в последующих частях программы можно использовать поля `FILE_OPEN` и т.д., как если бы они были константами, не опасаясь, что значение было изменено. Как показано в примере, общепринятое соглашение при написании кода предусматривает выбор для полей `final` идентификаторов со всеми буквами верхнего регистра.

Помимо полей как `final` могут быть объявлены и параметры метода, и локальные переменные. Объявление параметра как `final` предотвращает его изменение внутри метода. Объявление локальной переменной как `final` предотвращает присваивание ей значения более одного раза.

Ключевое слово `final` также может применяться к методам, но его смысл существенно отличается от того, когда оно применяется к переменным. Такое дополнительное использование `final` объясняется в следующей главе при описании наследования.

Снова о массивах

Массивы были представлены ранее в книге, еще до обсуждения классов. Теперь, когда вы ознакомились с классами, можно сделать важное замечание о массивах: они реализованы в виде объектов. По указанной причине массивы обладают особой характеристикой, которую вы захотите задействовать в своих интересах. В частности, размер массива, т.е. количество элементов, которые может содержать массив, находится в его переменной экземпляра `length`.

Все массивы имеют переменную `length`, и она всегда будет содержать размер массива. Ниже приведена программа, демонстрирующая эту характеристику:

```
// Демонстрация использования члена length в типе массива.
class Length {
  public static void main(String[] args) {
    int[] a1 = new int[10];
    int[] a2 = {3, 5, 7, 1, 8, 99, 44, -10};
    int[] a3 = {4, 3, 2, 1};
    System.out.println("Длина a1 равна " + a1.length);
    System.out.println("Длина a2 равна " + a2.length);
    System.out.println("Длина a3 равна " + a3.length);
  }
}
```

Программа генерирует следующий вывод:

```
Длина a1 равна 10
Длина a2 равна 8
Длина a3 равна 4
```

Как видите, в выводе отображается размер каждого массива. Имейте в виду, что значение `length` не имеет ничего общего с количеством фактически используемых элементов. Оно отражает только количество элементов, для хранения которых проектировался массив.

Вы можете найти хорошее применение члену `length` во многих ситуациях. Например, далее показана усовершенствованная версия класса `Stack`. Как вы помните, ранние версии класса `Stack` всегда создавали стек из десяти элементов. Новая версия позволяет создавать стеки любого размера. Переполнение стека предотвращается с применением значения `stck.length`.

```
// Усовершенствованный класс Stack, в котором
// используется член length в типе массива.
class Stack {
  private int[] stck;
  private int tos;
  // Разместить и инициализировать стек.
  Stack(int size) {
    stck = new int[size];
    tos = -1;
  }
  // Поместить элемент в стек.
  void push(int item) {
    if(tos==stck.length-1)  // использовать член length
      System.out.println("Стек полон.");
    else
      stck[++tos] = item;
  }
  // Извлечь элемент из стека.
  int pop() {
    if(tos < 0) {
      System.out.println("Стек опустошен.");
      return 0;
    }
```

```
    else
      return stck[tos--];
  }
}

class TestStack2 {
  public static void main(String[] args) {
    Stack mystack1 = new Stack(5);
    Stack mystack2 = new Stack(8);

    // Поместить несколько чисел в стеки.
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // Извлечь эти числа из стеков.
    System.out.println("Стек в mystack1:");
    for(int i=0; i<5; i++)
      System.out.println(mystack1.pop());

    System.out.println("Стек в mystack2:");
    for(int i=0; i<8; i++)
      System.out.println(mystack2.pop());
  }
}
```

Обратите внимание, что в программе создаются два стека: один рассчитан на хранение пяти, а другой — восьми элементов. Как видите, тот факт, что массивы поддерживают собственную информацию о длине, упрощает создание стеков любого размера.

Вложенные и внутренние классы

Класс можно определять внутри другого класса; такой класс известен как *вложенный класс*. Область действия вложенного класса ограничена областью действия его объемлющего класса. Таким образом, если класс B определен внутри класса A, то B не существует независимо от A. Вложенный класс имеет доступ к членам, в том числе закрытым, класса, в который он вложен. Тем не менее, объемлющий класс не имеет доступа к членам вложенного класса. Вложенный класс, объявленный непосредственно в области действия его объемлющего класса, будет членом объемлющего класса. Также можно объявлять вложенный класс, локальный для блока.

Существуют два типа вложенных классов: *статические* и *нестатические*. Статический вложенный класс — это класс, к которому применяется модификатор `static`. Поскольку класс статический, он должен обращаться к нестатическим членам объемлющего класса через объект. То есть статический вложенный класс не может напрямую ссылаться на нестатические члены объемлющего класса.

Вторым типом вложенного класса является *внутренний* класс. Внутренний класс — это нестатический вложенный класс. Он имеет доступ ко всем переменным и методам своего внешнего класса и может ссылаться на них напрямую так же, как поступают другие нестатические члены внешнего класса.

В следующей программе иллюстрируется определение и использование внутреннего класса. Класс по имени Outer имеет одну переменную экземпляра по имени external_x, один метод экземпляра с именем test() и определяет один внутренний класс по имени Inner.

```
// Демонстрация работы с внутренним классом.
class Outer {
  int outer_x = 100;
  void test() {
    Inner inner = new Inner();
    inner.display();
  }
  // Внутренний класс.
  class Inner {
    void display() {
      System.out.println("display(): outer_x = " + outer_x);
    }
  }
}
class InnerClassDemo {
  public static void main(String[] args) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Вот вывод, генерируемый программой:

```
display(): outer_x = 100
```

В программе внутренний класс по имени Inner определен в рамках области действия класса Outer, поэтому любой код класса Inner может напрямую обращаться к переменной external_x. В классе Inner определен метод экземпляра display(), который отображает external_x в стандартном потоке вывода. Метод main() объекта InnerClassDemo создает экземпляр класса Outer и вызывает его метод test(), который создает экземпляр класса Inner и вызывает метод display().

Важно понимать, что экземпляр Inner может быть создан только в контексте класса Outer. В противном случае компилятор Java сгенерирует сообщение об ошибке. Как правило, экземпляр внутреннего класса часто создается кодом в пределах его области действия, как сделано в примере.

Ранее уже объяснялось, что внутренний класс имеет доступ ко всем членам окружающего его класса, но обратное неверно. Члены внутреннего класса известны только в рамках области действия внутреннего класса и не могут использоваться внешним классом. Например:

```
// Эта программа не скомпилируется.
class Outer {
  int outer_x = 100;
  void test() {
    Inner inner = new Inner();
    inner.display();
  }
```

```
  // Внутренний класс.
  class Inner {
    int y = 10;              // переменная y является локальной для Inner

    void display() {
      System.out.println("display(): outer_x = " + outer_x);
    }
  }

  void showy() {
    System.out.println(y); // ошибка, переменная y здесь неизвестна!
  }
}

class InnerClassDemo {
  public static void main(String[] args) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Здесь y объявлена как переменная экземпляра Inner. Таким образом, за пределами этого класса она не известна и не может использоваться в методе showy().

Хотя мы сосредоточились на внутренних классах, объявленных в виде членов в области действия внешнего класса, внутренние классы можно определять в рамках области действия любого блока. Скажем, вложенный класс можно определить в блоке, который определяется методом, или даже в теле цикла for, как показано в следующей программе:

```
// Определение внутреннего класса в пределах цикла for.
class Outer {
  int outer_x = 100;

  void test() {
    for(int i=0; i<10; i++) {
      class Inner {
        void display() {
          System.out.println("display(): outer_x = " + outer_x);
        }
      }
      Inner inner = new Inner();
      inner.display();
    }
  }
}

class InnerClassDemo {
  public static void main(String[] args) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Ниже приведен вывод, генерируемый данной версией программы:

```
display(): outer_x = 100
display(): outer_x = 100
display(): outer_x = 100
display(): outer_x = 100
display(): outer_x = 100
display(): outer_x = 100
display(): outer_x = 100
display(): outer_x = 100
display(): outer_x = 100
display(): outer_x = 100
```

Хотя вложенные классы применимы не во всех ситуациях, они особенно полезны при обработке событий. Мы вернемся к теме вложенных классов в главе 25. Там вы увидите, как можно использовать внутренние классы с целью упрощения кода, необходимого для обработки определенных типов событий. Вы также узнаете об *анонимных внутренних классах*, которые представляют собой внутренние классы, не имеющие имени.

Интересно отметить, что вложенные классы не были разрешены в исходной спецификации Java 1.0, а появились в Java 1.1.

Исследование класса `String`

Несмотря на то что класс `String` будет подробно исследоваться в части II книги, его краткое изучение оправдано сейчас, потому что мы будем применять строки в ряде примеров программ, приведенных ближе к концу части I. Вероятно, `String` будет наиболее часто используемым классом в библиотеке классов Java. Очевидная причина связана с тем, что строки являются очень важной частью программирования.

Первое, что нужно понять касательно строк: каждая создаваемая вами строка на самом деле является объектом типа `String`. Даже строковые константы на самом деле представляют собой объекты `String`. Скажем, в следующем операторе строка `"Это тоже строка"` является объектом `String`:

```
System.out.println("Это тоже строка");
```

Второе, что важно понимать в отношении строк: объекты типа `String` неизменяемы; после создания объекта `String` его содержимое модифицировать нельзя. Хотя упомянутое ограничение может показаться серьезным, это не так по двум причинам.

- Если вам необходимо изменить строку, то вы всегда можете создать новую строку, отражающую изменения.
- В Java определены равноправные классы `String`, называемые `StringBuffer` и `StringBuilder`, которые позволяют изменять строки, так что в Java по-прежнему доступны все обычные операции со строками. (Классы `StringBuffer` и `StringBuilder` описаны в части II книги.)

Строки можно конструировать различными способами. Проще всего применять оператор такого вида:

```
String myString = "это просто тест";
```

После создания объект String можно использовать везде, где разрешена строка. Например, следующий оператор отображает myString:

```
System.out.println(myString);
```

Для объектов String в Java определена одна операция: +. Она применяется для конкатенации двух строк. Скажем, в результате выполнения показанного ниже оператора в myString будет содержаться строка "Мне нравится язык Java.":

```
String myString = "Мне" + " нравится " + "язык Java.";
```

Представленные выше концепции продемонстрированы в следующей программе:

```
// Демонстрация работы с объектами String.
class StringDemo {
  public static void main(String[] args) {
    String strOb1 = "Первая строка";
    String strOb2 = "Вторая строка";
    String strOb3 = strOb1 + " и " + strOb2;
    System.out.println(strOb1);

    System.out.println(strOb2);
    System.out.println(strOb3);
  }
}
```

Вот вывод, сгенерированный программой:

```
Первая строка
Вторая строка
Первая строка и Вторая строка
```

Класс String содержит несколько методов, которыми можно пользоваться. Рассмотрим несколько из них. Две строки можно проверить на предмет равенства с применением метода equals(). Вызвав метод length(), можно получить длину строки, а вызвав метод charAt() можно извлечь символ по указанному индексу в строке. Ниже показаны общие формы упомянутых трех методов:

```
boolean equals(secondStr)
int length()
char charAt(index)
```

А вот программа, в которой используются эти методы:

```
// Демонстрация работы нескольких методов класса String.
class StringDemo2 {
  public static void main(String[] args) {
    String strOb1 = "Первая строка";
    String strOb2 = "Вторая строка";
    String strOb3 = strOb1;

    System.out.println("Длина строки strOb1: " + strOb1.length());

    System.out.println("Символ по индексу 3 в строке strOb1: " +
                       strOb1.charAt(3));
```

```
    if(strOb1.equals(strOb2))
      System.out.println("Строка strOb1 равна строке strOb2");
    else
      System.out.println("Строка strOb1 не равна строке strOb2");

    if(strOb1.equals(strOb3))
      System.out.println("Строка strOb1 равна строке strOb3");
    else
      System.out.println("Строка strOb1 не равна строке strOb3");
  }
}
```

Программа генерирует следующий вывод:

```
Длина строки strOb1: 13
Символ по индексу 3 в строке strOb1: в
Строка strOb1 не равна строке strOb2
Строка strOb1 равна строке strOb3
```

Конечно же, можно создавать и массивы строк подобно массивам объектов любого другого типа. Например:

```
// Демонстрация использования массивов String.
class StringDemo3 {
  public static void main(String[] args) {
    String[] str = { "один", "два", "три" };

    for(int i=0; i<str.length; i++)
      System.out.println("str[" + i + "]: " +
                         str[i]);
  }
}
```

Вот вывод, генерируемый программой:

```
str[0]: один
str[1]: два
str[2]: три
```

В следующем разделе вы увидите, что массивы строк играют важную роль во многих программах на Java.

Использование аргументов командной строки

Иногда программе при запуске необходимо передать какую-то информацию. Для этого предназначены *аргументы командной строки*, передаваемые методу `main()`. Аргумент командной строки представляет собой данные, которые следуют непосредственно за именем программы в командной строке, когда программа запускается. Получить доступ к аргументам командной строки в программе на Java довольно просто — они хранятся в виде строк внутри массива типа `String`, который передается параметру `args` метода `main()`. Первый аргумент командной строки хранится в `args[0]`, второй — в `args[1]` и т.д. Приведенная далее программа отображает все аргументы командной строки, с которыми она вызывается:

```
// Отображение всех аргументов командной строки.
class CommandLine {
  public static void main(String[] args) {
    for(int i=0; i<args.length; i++)
      System.out.println("args[" + i + "]: " +
                         args[i]);
  }
}
```

Запустите программу как показано ниже:

```
java CommandLine это всего лишь тест 100 -1
```

Программа сгенерирует следующий вывод:

```
args[0]: это
args[1]: всего
args[2]: лишь
args[3]: тест
args[4]: 100
args[5]: -1
```

Помните! Все аргументы командной строки передаются в строковом виде. Как будет объяснено в главе 19, числовые значения придется вручную преобразовывать в их внутренние формы.

Аргументы переменной длины

В состав современных версий Java входит средство, упрощающее создание методов, которые должны принимать произвольное количество аргументов. Оно называется *аргументами переменной длины* (variable-length arguments — varargs). Метод, принимающий произвольное число аргументов, называется *методом с переменной арностью* или *методом с аргументами переменной длины.*

Ситуации, когда методу требуется передавать произвольное число аргументов, не являются чем-то необычным. Метод, который открывает подключение к Интернету, например, может принимать имя пользователя, пароль, имя файла, протокол и т.д., но предоставлять стандартные значения, если часть этой информации не была указана. В такой ситуации было бы удобно передавать только те аргументы, к которым не применяются стандартные значения. Другим примером может служить метод `printf()`, который является частью библиотеки ввода-вывода Java. Как вы увидите в главе 22, он принимает произвольное количество аргументов, форматирует их и затем отображает.

В ранних версиях Java аргументы переменной длины можно было поддерживать двумя способами, ни один из которых нельзя считать удобным. Первый способ, подходящий в ситуации, когда максимальное количество аргументов является небольшим и известным, предусматривал создание перегруженных версий метода, по одной для каждого варианта вызова метода. Хотя подобный подход работает и подходит в ряде случаев, он применим только к узкому набору ситуаций.

В тех случаях, когда максимальное количество потенциальных аргументов было большим или неизвестным, использовался второй подход, предусматривающий помещение аргументов в массив, который затем передавался методу. Данный подход все еще встречается в унаследованном коде и проиллюстрирован ниже:

```
// Использование массива для передачи методу произвольного числа аргументов.
// Это подход в старом стиле к аргументам переменной длины.
class PassArray {
  static void vaTest(int[] v) {
    System.out.print("Количество аргументов: " + v.length + " Содержимое: ");
    for(int x : v)
      System.out.print(x + " ");
    System.out.println();
  }

  public static void main(String[] args)
  {
    // Обратите внимание на то, как должен создаваться
    // массив для хранения аргументов.
    int[] n1 = { 10 };
    int[] n2 = { 1, 2, 3 };
    int[] n3 = { };

    vaTest(n1); // 1 аргумент
    vaTest(n2); // 3 аргумента
    vaTest(n3); // без аргументов
  }
}
```

Вывод, генерируемый программой, выглядит следующим образом:

```
Количество аргументов: 1 Содержимое: 10
Количество аргументов: 3 Содержимое: 1 2 3
Количество аргументов: 0 Содержимое:
```

Аргументы передаются методу `vaTest()` через массив `v`. Такой подход в старом стиле позволяет `vaTest()` принимать произвольное количество аргументов. Однако он требует, чтобы перед вызовом метода `vaTest()` аргументы вручную упаковывались в массив. Создание массива при каждом вызове `vaTest()` не только утомительно, но и потенциально подвержено ошибкам. Средство аргументов переменной длины предлагает более простой и совершенный вариант.

Аргумент переменной длины определяется с помощью трех точек (...). Например, вот как определить метод `vaTest()` с применением аргумента переменной длины:

```
static void vaTest(int ... v) {
```

Этот синтаксис сообщает компилятору о том, что метод `vaTest()` можно вызывать с нулем или большим числом аргументов. В результате `v` неявно объявляется как массив типа `int[]`. Таким образом, внутри `vaTest()` доступ к `v` осуществляется с использованием обычного синтаксиса массива. Ниже

показана предыдущая программа, переписанная с применением аргумента переменной длины:

```
// Демонстрация использования аргументов переменной длины.
class VarArgs {

  // vaTest() now uses a vararg.
  static void vaTest(int ... v) {
    System.out.print("Количество аргументов: " + v.length + " Содержимое: ");
    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String[] args)
  {
    // Обратите внимание, что теперь метод vaTest()
    // можно вызывать с переменным числом аргументов.
    vaTest(10);          // 1 аргумент
    vaTest(1, 2, 3);     // 3 аргумента
    vaTest();            // без аргументов
  }
}
```

Вывод, генерируемый программой, будет таким же, как в первоначальной версии.

В приведенной выше программе нужно отметить два важных момента. Во-первых, переменная `v` в методе `vaTest()` обрабатывается как массив. Дело в том, что `v` и *является* массивом. Синтаксис `...` просто сообщает компилятору, что будет использоваться переменное число аргументов, причем аргументы будут храниться в массиве, на который ссылается `v`. Во-вторых, метод `vaTest()` вызывается внутри `main()` с разным количеством аргументов, включая вариант вообще без аргументов. Аргументы автоматически помещаются в массив и передаются `v`. При отсутствии аргументов длина массива равна нулю.

Наряду с параметром переменной длины метод может иметь и "обычные" параметры. Тем не менее, параметр переменной длины должен объявляться в методе последним. Скажем, следующее объявление метода совершенно допустимо:

```
int doIt(int a, int b, double c, int ... vals) {
```

В этом случае первые три аргумента, указанные в вызове `doIt()`, сопоставляются с первыми тремя параметрами, а все остальные аргументы считаются относящимися к `vals`.

Не забывайте, что параметр переменной длины должен быть последним. Например, показанное далее объявление некорректно:

```
int doIt(int a, int b, double c, int...vals, boolean stopFlag) {//Ошибка!
```

Здесь предпринимается попытка объявить обычный параметр после параметра переменной длины, что недопустимо.

Существует еще одно ограничение, о котором следует помнить: должен быть только один параметр переменной длины. Скажем, приведенное ниже объявление тоже будет ошибочным:

```
int doIt(int a, int b, double c, int...vals,double...morevals) {//Ошибка!
```

Объявлять второй параметр переменной длины не разрешено.

Вот переработанная версия метода `vaTest()`, которая принимает обычный аргумент и аргумент переменной длины:

```
// Использование аргумента переменной длины со стандартными аргументами.
class VarArgs2 {
  //Здесь msg является нормальным параметром, а v - параметром переменной длины
  static void vaTest(String msg, int ... v) {
    System.out.print(msg + v.length + " Содержимое: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String[] args)
  {
    vaTest("Один аргумент переменной длины: ", 10);
    vaTest("Три аргумента переменной длины: ", 1, 2, 3);
    vaTest("Без аргументов переменной длины: ");
  }
}
```

Программа генерирует следующий вывод:

```
Один аргумент переменной длины: 1 Содержимое: 10
Три аргумента переменной длины: 3 Содержимое: 1 2 3
Без аргументов переменной длины: 0 Содержимое:
```

Перегрузка методов с аргументами переменной длины

Метод, принимающий аргумент переменной длины, можно перегружать. Например, в показанной далее программе метод `vaTest()` перегружается три раза:

```
// Аргументы переменной длины и перегрузка.
class VarArgs3 {

  static void vaTest(int ... v) {
    System.out.print("vaTest(int ...): " +
                     "Количество аргументов: " + v.length +
                     " Содержимое: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
                     "Количество аргументов: " + v.length + " Содержимое: ");
```

```
    for(boolean x : v)
      System.out.print(x + " ");
    System.out.println();
  }
  static void vaTest(String msg, int ... v) {
    System.out.print("vaTest(String, int ...): " +
                     msg + v.length +
                     " Содержимое: ");
    for(int x : v)
      System.out.print(x + " ");
    System.out.println();
  }
  public static void main(String[] args)
  {
    vaTest(1, 2, 3);
    vaTest("Тестирование: ", 10, 20);
    vaTest(true, false, false);
  }
}
```

Программа генерирует следующий вывод:

```
vaTest(int ...): Количество аргументов: 3 Содержимое: 1 2 3
vaTest(String, int ...): Тестирование: 2 Содержимое: 10 20
vaTest(boolean ...) Количество аргументов: 3 Содержимое: true false false
```

В представленной выше программе иллюстрируются оба способа перегрузки метода с аргументом переменной длины. Первый способ предусматривает применение отличающегося типа для параметра переменной длины, как в случае `vaTest(int ...)` и `vaTest(boolean ...)`. Вспомните, что `...` приводит к тому, что параметр интерпретируется в виде массива заданного типа. Следовательно, точно так же, как обычные методы можно перегружать с использованием отличающихся параметров типа массивов, методы с параметрами переменной длины разрешено перегружать, указывая разные типы для параметров переменной длины. В этом случае компилятор Java вызывает надлежащий перегруженный метод на основе отличия между типами.

Второй способ перегрузки метода с аргументом переменной длины предполагает добавление одного или нескольких обычных параметров, что и было сделано в версии `vaTest(String, int ...)`. В данном случае компилятор Java вызывает надлежащий перегруженный метод на основе и количества, и типа аргументов.

На заметку! Метод с аргументом переменной длины также может быть перегружен за счет определения метода без аргумента переменной длины. Скажем, в приведенной выше программе `vaTest(int x)` является допустимой версией `vaTest()`. Эта версия вызывается только при наличии одного аргумента типа `int`. Когда передаются два или более аргументов типа `int`, используется версия `vaTest(int ... v)` с аргументом переменной длины.

Аргументы переменной длины и неоднозначность

При перегрузке метода, принимающего аргумент переменной длины, могут возникать несколько неожиданные ошибки. Такие ошибки связаны с неоднозначностью, поскольку существует возможность создать двусмысленный вызов перегруженного метода с аргументами переменной длины. Например, взгляните на следующую программу:

```
// Аргументы переменной длины, перегрузка и неоднозначность.
//
// Эта программа содержит ошибку и не скомпилируется!
class VarArgs4 {

  static void vaTest(int ... v) {
    System.out.print("vaTest(int ...): " +
                     "Количество аргументов: " + v.length +
                     " Содержимое: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
                     "Количество аргументов: " + v.length +
                     " Содержимое: ");

    for(boolean x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String[] args)
  {
    vaTest(1, 2, 3);              // Нормально
    vaTest(true, false, false);   // Нормально
    vaTest();                     // Ошибка: Неоднозначность!
  }
}
```

В этой программе перегрузка метода `vaTest()` совершенно корректна, но программа не скомпилируется из-за такого вызова:

```
vaTest(); // Ошибка: Неоднозначность!
```

Поскольку параметр переменной длины может быть пустым, вызов будет транслироваться в `vaTest(int ...)` или в `vaTest(boolean ...)`, которые оба одинаково действительны. Таким образом, вызов в своей основе неоднозначен.

Ниже приведен еще один пример неоднозначности. Следующие перегруженные версии метода `vaTest()` по своей сути неоднозначны, хотя одна из них принимает обычный параметр:

```
static void vaTest(int ... v) {          // ...
static void vaTest(int n, int ... v) {   // ...
```

Несмотря на различие в списках параметров `vaTest()`, компилятор не сможет распознать следующий вызов:

```
vaTest(1)
```

Во что преобразуется данный вызов: в `vaTest(int ...)` с одним аргументом переменной длины или в `vaTest(int, int ...)` без аргументов переменной длины? Компилятор никак не сможет получить ответ на этот вопрос. Таким образом, ситуация неоднозначна.

Из-за ошибок неоднозначности, подобных только что показанным, иногда вам придется отказаться от перегрузки и просто использовать два метода с разными именами. Кроме того, в ряде случаев ошибки неоднозначности выявляют концептуальный дефект в коде, который можно устранить, более тщательно проработав решение.

Выведение типов локальных переменных для ссылочных типов

Как упоминалось в главе 3, начиная с версии JDK 10, в Java поддерживается выведение типов локальных переменных. Вспомните, что при выведении типов локальных переменных тип переменной указывается как `var`, а переменная должна быть инициализирована. В более ранних примерах выведение типов демонстрировалось с примитивными типами, но его также можно применять со ссылочными типами. Фактически основное использование выведения типов связано со ссылочными типами. Вот простой пример, в котором объявляется строковая переменная по имени `myStr`:

```
var myStr = "Это строка";
```

Из-за использования в качестве инициализатора строки в кавычках для переменной `myStr` выводится тип `String`.

Как объяснялось в главе 3, одно из преимуществ выведения типов локальных переменных связано с его способностью оптимизировать код, и такая оптимизация наиболее очевидна именно со ссылочными типами. Причина в том, что многие типы классов в Java имеют довольно длинные имена. Например, в главе 13 вы узнаете о классе `FileInputStream`, с помощью которого файл открывается для операций ввода. В прошлом объект `FileInputStream` объявлялся и инициализировался с применением традиционного объявления вроде показанного ниже:

```
FileInputStream fin = new FileInputStream("test.txt");
```

С использованием `var` теперь его можно переписать так:

```
var fin = new FileInputStream("test.txt");
```

Тут предполагается, что переменная `fin` имеет тип `FileInputStream`, т.к. это тип ее инициализатора. Нет никакой необходимости явно повторять имя типа. В результате такое объявление `fin` значительно короче, чем его запись традиционным способом, а потому применение `var` упрощает объявление.

Это преимущество становится еще более очевидным в более сложных объявлениях, например, включающих обобщения. В целом упрощение выведения типов локальных переменных помогает уменьшить утомительный набор длинных имен типов в программах.

Разумеется, аспект упрощения кода со стороны выведения типов локальных переменных должен использоваться осмотрительно, чтобы не ухудшить читабельность программы и в итоге не скрыть ее намерения. Например, взгляните на объявление следующего вида:

```
var x = o.getNext();
```

В данном случае кому-то, читающему ваш код, может быть не сразу станет ясно, какой тип имеет x. По существу выведение типов локальных переменных представляет собой средство, которое должно применяться обдуманно.

Выведение типов локальных переменных можно также использовать в отношении пользовательских классов, как демонстрируется в показанной ниже программе. В ней создается класс по имени `MyClass`, а затем с помощью выведения типов локальных переменных объявляется и инициализируется объект этого класса.

```
// Использование выведения типов локальных переменных
// с пользовательским классом.
class MyClass {
  private int i;

  MyClass(int k) { i = k;}

  int geti() { return i; }
  void seti(int k) { if(k >= 0) i = k; }
}

class RefVarDemo {
  public static void main(String[] args) {
    var mc = new MyClass(10); // Обратите внимание на применение var.

    System.out.println("Значение i в mc теперь равно " + mc.geti());
    mc.seti(19);
    System.out.println("Значение i в mc теперь равно " + mc.geti());
  }
}
```

Вот вывод, генерируемый программой:

```
Значение i в mc теперь равно 10
Значение i в mc теперь равно 19
```

Обратите особое внимание в программе на следующую строку:

```
var mc = new MyClass(10);     // Обратите внимание на применение var.
```

Тип переменной `mc` будет выводиться как `MyClass`, потому что это тип инициализатора, который является новым объектом `MyClass`.

Как объяснялось ранее в книге, для удобства читателей, работающих в средах Java, которые не поддерживают выведение типов локальных переменных, в большинстве примеров настоящего издания оно применяться не будет, что позволит их компилировать и запускать максимальному числу читателей.

ГЛАВА

8 Наследование

Наследование является одним из краеугольных камней объектно-ориентированного программирования (ООП), поскольку позволяет создавать иерархические классификации. С использованием наследования вы можете создать универсальный класс, который определяет характерные черты, общие для набора связанных элементов. Затем этот класс может быть унаследован другими, более специфическими классами, каждый из которых добавляет те элементы, которые уникальны для него. В терминологии Java унаследованный класс называется *суперклассом*, а класс, выполняющий наследование — *подклассом*. Следовательно, подкласс представляет собой специализированную версию суперкласса. Он наследует все члены, определенные суперклассом, и добавляет собственные уникальные элементы.

Основы наследования

Чтобы наследовать класс, вы просто включаете определение одного класса в другой с применением ключевого слова `extends`. Давайте выясним, как это делать, начав с простого примера. В следующей программе создается суперкласс по имени `A` и подкласс по имени `B`. Обратите внимание на использование ключевого слова `extends` для создания подкласса `A`.

```
// Простой пример наследования.

// Создать суперкласс.
class A {
  int i, j;

  void showij() {
    System.out.println("i и j: " + i + " " + j);
  }
}

// Создать подкласс путем расширения класса A.
class B extends A {
  int k;

  void showk() {
    System.out.println("k: " + k);
  }
```

```
  void sum() {
    System.out.println("i+j+k: " + (i+j+k));
  }
}

class SimpleInheritance {
  public static void main(String[] args) {
    A superOb = new A();
    B subOb = new B();

    // Суперкласс можно использовать сам по себе.
    superOb.i = 10;
    superOb.j = 20;
    System.out.println("Содержимое superOb: ");
    superOb.showij();
    System.out.println();

    /* Подкласс имеет доступ ко всем открытым членам своего суперкласса. */
    subOb.i = 7;
    subOb.j = 8;
    subOb.k = 9;
    System.out.println("Содержимое subOb: ");
    subOb.showij();
    subOb.showk();
    System.out.println();

    System.out.println("Сумма i, j и k в subOb:");
    subOb.sum();
  }
}
```

Ниже показан вывод, генерируемый программой:

```
Содержимое superOb:
i и j: 10 20

Содержимое subOb:
i и j: 7 8
k: 9

Сумма i, j и k в subOb:
i+j+k: 24
```

Как видите, подкласс B включает в себя все члены своего суперкласса A. Вот почему объект subOb может получать доступ к i и j и вызывать showij(). Кроме того, внутри sum() на i и j можно ссылаться напрямую, как если бы они были частью B.

Несмотря на то что A выступает в качестве суперкласса для B, он также является полностью независимым, автономным классом. Быть суперклассом для подкласса не означает, что суперкласс не может использоваться сам по себе. Кроме того, подкласс может быть суперклассом для другого подкласса.

Общая форма объявления класса, унаследованного от суперкласса, выглядит следующим образом:

```
class имя-подкласса extends имя-суперкласса {
  // тело класса
}
```

Для любого создаваемого подкласса разрешено указывать только один суперкласс. Наследование нескольких суперклассов при создании одиночного подкласса в языке Java не поддерживается. Как было указано, можно создать иерархию наследования, в которой подкласс становится суперклассом для другого подкласса. Однако ни один класс не может быть суперклассом для самого себя.

Доступ к членам и наследование

Хотя подкласс включает в себя все члены своего суперкласса, он не может получить доступ к тем членам суперкласса, которые были объявлены как закрытые. Например, рассмотрим приведенную далее простую иерархию классов:

```
/* В иерархии классов члены private остаются закрытыми
   по отношению к своему классу.

   Эта программа содержит ошибку и не скомпилируется.
*/

// Создать суперкласс.
class A {
  int i;                  // стандартный доступ
  private int j;          // закрыт по отношению к A

  void setij(int x, int y) {
    i = x;
    j = y;
  }
}

// Член j из класса A здесь недоступен.
class B extends A {
  int total;

  void sum() {
    total = i + j;        // ОШИБКА, член j здесь недоступен
  }
}

class Access {
  public static void main(String[] args) {
    B subOb = new B();

    subOb.setij(10, 12);

    subOb.sum();
    System.out.println("Сумма равна " + subOb.total);
  }
}
```

Программа не скомпилируется, потому что использование `j` внутри метода `sum()` класса `B` вызывает нарушение прав доступа. Поскольку член `j` объявлен как `private`, он доступен только другим членам собственного класса. Подклассы не имеют к нему доступа.

Помните! Член класса, который был объявлен как `private`, останется закрытым по отношению к своему классу. Он не будет доступен любому коду за пределами своего класса, в том числе подклассам.

Более реалистичный пример

Давайте рассмотрим более реалистичный пример, который поможет оценить всю мощь наследования. Здесь финальная версия класса `Box`, разработанная в предыдущей главе, будет расширена за счет включения четвертого компонента по имени `weight` (вес). Таким образом, новый класс будет содержать ширину, высоту, глубину и вес коробки.

```
// В этой программе используется наследование для расширения класса Box.
class Box {
  double width;
  double height;
  double depth;

  // Конструктор, применяемый для клонирования объекта.
  Box(Box ob) {  // передать объект конструктору
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // Конструктор, используемый в случае указания всех размеров.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // Конструктор, применяемый в случае, если размеры вообще не указаны.
  Box() {
    width = -1;      // использовать -1 для обозначения
    height = -1;     // неинициализированного
    depth = -1;      // объекта коробки
  }

  // Конструктор, используемый в случае создания объекта кубической
коробки.
  Box(double len) {
    width = height = depth = len;
  }

  // Вычислить и возвратить объем.
  double volume() {
    return width * height * depth;
  }
}

// Здесь класс Box расширяется с целью включения члена weight.
class BoxWeight extends Box {
  double weight;     // вес коробки

  // Конструктор для BoxWeight.
  BoxWeight(double w, double h, double d, double m) {
```

```
    width = w;
    height = h;
    depth = d;
    weight = m;
  }
}
class DemoBoxWeight {
  public static void main(String[] args) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    double vol;

    vol = mybox1.volume();
    System.out.println("Объем mybox1 равен " + vol);
    System.out.println("Вес mybox1 равен " + mybox1.weight);
    System.out.println();

    vol = mybox2.volume();
    System.out.println("Объем mybox2 равен " + vol);
    System.out.println("Вес mybox2 равен " + mybox2.weight);
  }
}
```

Вот вывод, генерируемый программой:

```
Объем mybox1 равен 3000.0
Вес mybox1 равен 34.3

Объем mybox2 равен 24.0
Вес mybox2 равен 0.076
```

Класс `BoxWeight` наследует все характеристики класса `Box` и добавляет к ним компонент `weight` для представления веса. Классу `BoxWeight` вовсе не обязательно воссоздавать все функциональные средства, имеющиеся в `Box`. Для достижения своих целей он может просто расширить `Box`.

Основное преимущество наследования связано с тем, что после создания суперкласса, который определяет характерные черты, общие для набора объектов, его можно применять для создания любого количества более конкретных подклассов. Каждый подкласс может точно настраивать собственное предназначение. Скажем, следующий класс наследуется от `Box` и добавляет свойство цвета:

```
// Здесь класс Box расширяется для включения свойства цвета.
class ColorBox extends Box {
  int color;  // цвет коробки

  ColorBox(double w, double h, double d, int c) {
    width = w;
    height = h;
    depth = d;
    color = c;
  }
}
```

Не забывайте, что после создания суперкласса, определяющего общие аспекты объекта, данный суперкласс можно наследовать для формирования

специализированных классов. Каждый подкласс просто добавляет свои уникальные характеристики. В этом и заключается суть наследования.

Переменная типа суперкласса может ссылаться на объект подкласса

Ссылочной переменной типа суперкласса можно присваивать ссылку на объект любого подкласса, производного от данного суперкласса. Вы сочтете такой аспект наследования весьма полезным в разнообразных ситуациях. Рассмотрим показанный ниже код примера:

```
class RefDemo {
  public static void main(String[] args) {
    BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
    Box plainbox = new Box();
    double vol;

    vol = weightbox.volume();
    System.out.println("Объем weightbox равен " + vol);
    System.out.println("Вес weightbox равен " +
                       weightbox.weight);
    System.out.println();

    // Присвоить ссылку на BoxWeight ссылке на Box.
    plainbox = weightbox;

    vol = plainbox.volume();  // нормально, метод volume() определен в Box
    System.out.println("Объем plainbox равен " + vol);

    /* Следующий оператор ошибочен, потому что член weight в plainbox
       не определен. */
    // System.out.println("Вес plainbox равен " + plainbox.weight);
  }
}
```

Здесь `weightbox` является ссылкой на объекты `BoxWeight`, а `plainbox` — ссылкой на объекты `Box`. Поскольку `BoxWeight` — подкласс `Box`, переменной `plainbox` разрешено присваивать ссылку на объект `weightbox`.

Важно понимать, что именно тип ссылочной переменной, а не тип объекта, на который она ссылается, определяет, к каким членам можно получать доступ. Другими словами, когда ссылочной переменной типа суперкласса присваивается ссылка на объект подкласса, то доступ имеется только к тем частям объекта, которые определены в суперклассе. Вот почему переменная `plainbox` не может получить доступ к `weight`, даже если она ссылается на объект `BoxWeight`. Если подумать, то в этом есть смысл, потому что суперклассу ничего не известно о том, *что* к нему добавляет подкласс. Поэтому последняя строка кода в предыдущем фрагменте закомментирована. Ссылка `Box` не может получить доступ к полю `weight`, т.к. в классе `Box` оно не определено.

Хотя описанный выше прием может показаться несколько экзотическим, с ним связан ряд важных практических применений, два из которых обсуждаются далее в главе.

Использование ключевого слова super

В предшествующих примерах классы, производные от Box, не были реализованы настолько эффективно и надежно, насколько могли бы. Скажем, конструктор для BoxWeight явно инициализирует поля width, height и depth класса Box. Это не только приводит к дублированию кода, уже имеющегося в его суперклассе, что неэффективно, но и подразумевает предоставление подклассу доступа к упомянутым членам. Тем не менее, будут возникать ситуации, когда желательно создавать суперкласс, который держит детали своей реализации при себе (т.е. хранит свои элементы данных закрытыми). В таком случае у подкласса не было бы возможности напрямую обращаться к этим переменным либо инициализировать их самостоятельно. Поскольку инкапсуляция является основным атрибутом ООП, совершенно не удивительно, что в Java предлагается решение описанной проблемы. Всякий раз, когда подклассу необходимо сослаться на свой непосредственный суперкласс, он может воспользоваться ключевым словом super.

Ключевое слово super имеет две основные формы. Первая вызывает конструктор суперкласса, а вторая служит для доступа к члену суперкласса, который был сокрыт членом подкласса. Обе формы обсуждаются далее в главе.

Использование ключевого слова super для вызова конструкторов суперкласса

Подкласс может вызывать конструктор, определенный в его суперклассе, с применением следующей формы super:

```
super(список-аргументов);
```

Здесь список-аргументов предназначен для указания любых аргументов, необходимых конструктору в суперклассе. Вызов super() всегда должен быть первым оператором, выполняемым внутри конструктора подкласса.

Чтобы увидеть, как используется super(), рассмотрим показанную ниже усовершенствованную версию класса BoxWeight:

```
// В классе BoxWeight члены его суперкласса Box теперь инициализируются с применением super.
class BoxWeight extends Box {
  double weight;    // вес коробки

  // Инициализировать width, height и depth, используя super().
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d);  // вызвать конструктор суперкласса
    weight = m;
  }
}
```

Конструктор BoxWeight() вызывает super() с аргументами w, h и d, что приводит к вызову конструктора Box, который инициализирует поля width, height и depth с применением этих значений. Класс BoxWeight больше не инициализирует указанные поля самостоятельно. Ему нужно инициализиро-

вать только уникальное для него поле: weight. Таким образом, появляется возможность при желании сделать поля width, height и depth в классе Box закрытыми.

В предыдущем примере вызов super() содержал три аргумента. Поскольку конструкторы могут быть перегружены, super() можно вызывать с использованием любой формы, определенной в суперклассе. Выполнится тот конструктор, который дает совпадение по аргументам. Например, далее представлена законченная реализация BoxWeight, предлагающая конструкторы для различных способов создания объекта коробки. В каждом случае super() вызывается с применением подходящих аргументов. Обратите внимание, что поля width, height и depth в классе Box сделаны закрытыми.

```
// Законченная реализация класса BoxWeight.
class Box {
  private double width;
  private double height;
  private double depth;

  // Конструктор, применяемый для клонирования объекта.
  Box(Box ob) {      // передать объект конструктору
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // Конструктор, используемый в случае указания всех размеров.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // Конструктор, применяемый в случае, если размеры вообще не указаны.
  Box() {
    width = -1;      // использовать -1 для обозначения
    height = -1;     // неинициализированного
    depth = -1;      // объекта коробки
  }

  // Конструктор, используемый в случае создания объекта
  // кубической коробки.
  Box(double len) {
    width = height = depth = len;
  }

  // Вычислить и возвратить объем.
  double volume() {
    return width * height * depth;
  }
}

// Теперь класс BoxWeight полностью реализует все конструкторы.
class BoxWeight extends Box {
  double weight;     // вес коробки
```

```
  // Конструктор, применяемый для клонирования объекта.
  BoxWeight(BoxWeight ob) { // передать объект конструктору
    super(ob);
    weight = ob.weight;
  }

  // Конструктор, используемый в случае указания всех параметров.
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d);           // вызвать конструктор суперкласса
    weight = m;
  }

  // Стандартный конструктор.
  BoxWeight() {
    super();
    weight = -1;
  }

  // Конструктор, используемый в случае создания объекта кубической коробки
  BoxWeight(double len, double m) {
    super(len);
    weight = m;
  }
}
class DemoSuper {
  public static void main(String[] args) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    BoxWeight mybox3 = new BoxWeight(); // стандартный
    BoxWeight mycube = new BoxWeight(3, 2);
    BoxWeight myclone = new BoxWeight(mybox1);
    double vol;

    vol = mybox1.volume();
    System.out.println("Объем mybox1 равен " + vol);
    System.out.println("Вес mybox1 равен " + mybox1.weight);
    System.out.println();

    vol = mybox2.volume();
    System.out.println("Объем mybox2 равен " + vol);
    System.out.println("Вес mybox2 равен " + mybox2.weight);
    System.out.println();

    vol = mybox3.volume();
    System.out.println("Объем mybox3 равен " + vol);
    System.out.println("Вес mybox3 равен " + mybox3.weight);
    System.out.println();

    vol = myclone.volume();
    System.out.println("Объем myclone равен " + vol);
    System.out.println("Вес myclone равен " + myclone.weight);
    System.out.println();

    vol = mycube.volume();
    System.out.println("Объем mycube равен " + vol);
    System.out.println("Вес mycube равен " + mycube.weight);
    System.out.println();
  }
}
```

Вот вывод, генерируемый программой:

```
Объем mybox1 равен 3000.0
Вес mybox1 равен 34.3

Объем mybox2 равен 24.0
Вес mybox2 равен 0.076

Объем mybox3 равен -1.0
Вес mybox3 равен -1.0

Объем myclone равен 3000.0
Вес myclone равен 34.3

Объем mycube равен 27.0
Вес mycube равен 2.0
```

Обратите особое внимание на следующий конструктор в `BoxWeight`:

```
// Конструктор, применяемый для клонирования объекта.
BoxWeight(BoxWeight ob) {  // передать объект конструктору
  super(ob);
  weight = ob.weight;
}
```

Здесь вызову `super()` передается объект типа `BoxWeight`, а не `Box`. По-прежнему вызывается конструктор `Box(Box ob)`. Как упоминалось ранее, переменная типа суперкласса может использоваться для ссылки на любой объект, производный от этого класса, т.е. мы можем передавать конструктору `Box` объект `BoxWeight`. Конечно же, классу `Box` известны только свои члены.

Давайте повторим ключевые концепции, лежащие в основе `super()`. Когда подкласс вызывает `super()`, он вызывает конструктор своего непосредственного суперкласса. Таким образом, `super()` всегда ссылается на суперкласс непосредственно над вызывающим классом. Это справедливо даже для многоуровневой иерархии. Кроме того, вызов `super()` всегда должен быть первым оператором, выполняемым внутри конструктора подкласса.

Использование второй формы ключевого слова `super`

Вторая форма ключевого слова `super` действует примерно так же, за исключением того, что всегда относится к суперклассу подкласса, в котором задействована. Вот как она выглядит:

```
super.член
```

Здесь член может быть либо методом, либо переменной экземпляра.

Вторая форма `super` наиболее применима в ситуациях, когда имена членов подкласса скрывают члены с тем же именем в суперклассе. Возьмем следующую простую иерархию классов:

```
// Использование super для преодоления сокрытия имен.
class A {
  int i;
}
```

```
// Создать подкласс путем расширения класса A.
class B extends A {
  int i;             // этот член i скрывает i в A

  B(int a, int b) {
    super.i = a;     // i в A
    i = b;           // i в B
  }

  void show() {
    System.out.println("i в суперклассе: " + super.i);
    System.out.println("i в подклассе: " + i);
  }
}

class UseSuper {
  public static void main(String[] args) {
    B subOb = new B(1, 2);

    subOb.show();
  }
}
```

Ниже показан вывод, генерируемый программой:

```
i в суперклассе: 1
i в подклассе: 2
```

Хотя переменная экземпляра `i` в `B` скрывает `i` в `A`, ключевое слово `super` делает возможным доступ к члену `i`, определенному в суперклассе. Как вы увидите, `super` можно также использовать для вызова методов, сокрытых подклассом.

Создание многоуровневой иерархии

До сих пор мы имели дело с простыми иерархиями классов, которые состояли только из суперкласса и подкласса. Однако можно создавать иерархии, содержащие любое количество уровней наследования. Как уже упоминалось, подкласс вполне допустимо применять в качестве суперкласса другого. Например, при наличии трех классов `A`, `B` и `C` класс `C` может быть подклассом `B`, который является подклассом `A`. Когда возникает ситуация такого типа, каждый подкласс наследует все характерные черты, обнаруженные во всех его суперклассах. В данном случае `C` наследует все аспекты `B` и `A`. Чтобы увидеть, чем может быть полезна многоуровневая иерархия, рассмотрим следующую программу, где подкласс `BoxWeight` используется как суперкласс для создания подкласса `Shipment`, представляющего доставку коробки. Класс `Shipment` наследует все признаки `BoxWeight` и `Box` и добавляет поле по имени `cost`, в котором содержится стоимость доставки такой посылки.

```
// Расширение класса BoxWeight с целью включения стоимости доставки.

// Начать с Box.
class Box {
  private double width;
  private double height;
```

```
  private double depth;
  // Конструктор, применяемый для клонирования объекта.
  Box(Box ob) {               // передать объект конструктору
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }
  // Конструктор, используемый в случае указания всех размеров.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
  // Конструктор, применяемый в случае, если размеры вообще не указаны.
  Box() {
    width = -1;               // использовать -1 для обозначения
    height = -1;              // неинициализированного
    depth = -1;               // объекта коробки
  }
  // Конструктор, используемый в случае создания объекта кубической коробки
  Box(double len) {
    width = height = depth = len;
  }
  // Вычислить и возвратить объем.
  double volume() {
    return width * height * depth;
  }
}
// Добавить вес.
class BoxWeight extends Box {
  double weight;              // вес коробки
  // Конструктор, применяемый для клонирования объекта.
  BoxWeight(BoxWeight ob) {   // передать объект конструктору
    super(ob);
    weight = ob.weight;
  }
  // Конструктор, используемый в случае указания всех параметров.
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d);           // вызвать конструктор суперкласса
    weight = m;
  }
  // Стандартный конструктор.
  BoxWeight() {
    super();
    weight = -1;
  }
  // Конструктор, используемый в случае создания объекта кубической коробки
  BoxWeight(double len, double m) {
    super(len);
    weight = m;
  }
}
```

```
// Добавить стоимость доставки.
class Shipment extends BoxWeight {
  double cost;

  // Конструктор, применяемый для клонирования объекта.
  Shipment(Shipment ob) {    // передать объект конструктору
    super(ob);
    cost = ob.cost;
  }

  // Конструктор, используемый в случае указания всех параметров.
  Shipment(double w, double h, double d,
           double m, double c) {
    super(w, h, d, m);        // вызвать конструктор суперкласса
    cost = c;
  }

  // Стандартный конструктор.
  Shipment() {
    super();
    cost = -1;
  }

  // Конструктор, используемый в случае создания объекта кубической коробки
  Shipment(double len, double m, double c) {
    super(len, m);
    cost = c;
  }
}

class DemoShipment {
  public static void main(String[] args) {
    Shipment shipment1 = new Shipment(10, 20, 15, 10, 3.41);
    Shipment shipment2 = new Shipment(2, 3, 4, 0.76, 1.28);
    double vol;

    vol = shipment1.volume();
    System.out.println("Объем shipment1 равен " + vol);
    System.out.println("Вес shipment1 равен " + shipment1.weight);
    System.out.println("Стоимость доставки: $" + shipment1.cost);
    System.out.println();

    vol = shipment2.volume();
    System.out.println("Объем shipment2 равен " + vol);
    System.out.println("Вес shipment2 равен " + shipment2.weight);
    System.out.println("Стоимость доставки: $" + shipment2.cost);
  }
}
```

Вот вывод, генерируемый программой:

```
Объем shipment1 равен 3000.0
Вес shipment1 равен 10.0
Стоимость доставки: $3.41

Объем shipment2 равен 24.0
Вес shipment2 равен 0.76
Стоимость доставки: $1.28
```

Благодаря наследованию класс Shipment может задействовать определенные ранее классы Box и BoxWeight, добавляя только ту дополнительную информацию, которая необходима для собственного конкретного приложения. Это часть ценности наследования; оно позволяет многократно использовать код.

В примере иллюстрируется еще один важный момент: super() всегда ссылается на конструктор в ближайшем суперклассе. В классе Shipment с помощью super() вызывается конструктор BoxWeight, а в классе BoxWeight — конструктор Box. В рамках иерархии классов, когда конструктору суперкласса требуются аргументы, то все подклассы должны передавать их "вверх по цепочке наследования". Сказанное верно независимо от того, нужны ли подклассу собственные аргументы.

На заметку! В предыдущей программе вся иерархия классов, включая Box, BoxWeight и Shipment, находится в одном файле, что служит только ради удобства. В Java все три класса можно было бы поместить в собственные файлы и скомпилировать по отдельности. На самом деле при создании иерархий классов использование отдельных файлов является нормой, а не исключением.

Когда конструкторы выполняются

Когда иерархия классов создана, в каком порядке выполняются конструкторы классов, образующих иерархию? Например, при наличии подкласса B и суперкласса A конструктор A выполняется раньше конструктора B или наоборот? Ответ заключается в том, что в иерархии классов конструкторы завершают свое выполнение в порядке наследования от суперкласса к подклассу. Кроме того, поскольку вызов super() должен быть первым оператором, выполняемым в конструкторе подкласса, такой порядок остается тем же независимо от того, применяется super() или нет. Если super() не используется, то будет выполнен стандартный конструктор или конструктор без параметров каждого суперкласса. Выполнение конструкторов демонстрируется в следующей программе:

```
// Демонстрация выполнения конструкторов.

// Создать суперкласс.
class A {
  A() {
    System.out.println("Внутри конструктора A.");
  }
}

// Создать подкласс путем расширения класса A.
class B extends A {
  B() {
    System.out.println("Внутри конструктора B.");
  }
}

// Создать еще один подкласс путем расширения класса B.
class C extends B {
```

```
  C() {
    System.out.println("Внутри конструктора C.");
  }
}

class CallingCons {
  public static void main(String[] args) {
    C c = new C();
  }
}
```

Вот вывод, генерируемый программой:

```
Внутри конструктора A
Внутри конструктора B
Внутри конструктора C
```

Как видите, конструкторы выполняются в порядке наследования.

Если хорошо подумать, то имеет смысл, что конструкторы завершают свое выполнение в порядке наследования. Поскольку суперклассу ничего не известно о каких-либо подклассах, любая инициализация, которую должен выполнить суперкласс, является отдельной и возможно обязательной для любой инициализации, выполняемой подклассом. Следовательно, она должна быть завершена первой.

Переопределение методов

В иерархии классов, когда метод в подклассе имеет то же имя и сигнатуру типа, что и метод в его суперклассе, то говорят, что метод в подклассе *переопределяет* метод в суперклассе. При вызове переопределенного метода через его подкласс всегда будет вызываться версия метода, определенная в подклассе. Версия метода, определенная в суперклассе, будет сокрыта. Рассмотрим следующий пример:

```
// Переопределение методов.
class A {
  int i, j;

  A(int a, int b) {
    i = a;
    j = b;
  }

  // Отобразить значения i и j.
  void show() {
    System.out.println("i и j: " + i + " " + j);
  }
}

class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }
```

```
  // Отобразить k - переопределяет show() в A.
  void show() {
    System.out.println("k: " + k);
  }
}
class Override {
  public static void main(String[] args) {
    B subOb = new B(1, 2, 3);
    subOb.show(); // вызывается show() из B
  }
}
```

Ниже показан вывод, генерируемый программой:

```
k: 3
```

Когда метод `show()` вызывается на объекте типа `B`, используется версия `show()`, определенная в классе `B`. То есть версия `show()` внутри `B` переопределяет версию `show()`, объявленную в `A`.

При желании получить доступ к версии переопределенного метода из суперкласса можно применить ключевое слово `super`. Скажем, в приведенном далее классе `B` внутри версии `show()` из подкласса вызывается версия `show()` из суперкласса, что позволяет отобразить все переменные экземпляра.

```
class B extends A {
  int k;
  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }
  void show() {
    super.show(); // вызывается show() из A
    System.out.println("k: " + k);
  }
}
```

Подставив эту версию класса `B` в код предыдущего примера, вы увидите такой вывод:

```
i и j: 1 2
k: 3
```

Здесь `super.show()` вызывает версию `show()` из суперкласса.

Метод переопределяется *только* в случае, если имена и сигнатуры типов двух методов идентичны, а иначе два метода будут просто перегруженными. Взгляните на следующую модифицированную версию предыдущего примера:

```
// Методы с отличающимися сигнатурами типов
// являются перегруженными - не переопределенными.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }
```

```
  // Отобразить значения i и j.
  void show() {
    System.out.println("i и j: " + i + " " + j);
  }
}

// Создать подкласс путем расширения класса A.
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // Перегрузить show().
  void show(String msg) {
    System.out.println(msg + k);
  }
}

class Override {
  public static void main(String[] args) {
    B subOb = new B(1, 2, 3);

    subOb.show("Это k: ");    // вызывается show() из B
    subOb.show();             // вызывается show() из A
  }
}
```

Вот вывод, генерируемый программой:

```
Это k: 3
i и j: 1 2
```

Версия метода `show()` в классе `B` принимает строковый параметр, что отличает его сигнатуру типов от сигнатуры метода `show()` в классе `A`, который не принимает параметров. Поэтому никакого переопределения (или сокрытия имени) не происходит. Взамен версия `show()` в классе `B` просто перегружает версию `show()` из класса `A`.

Динамическая диспетчеризация методов

Хотя примеры в предыдущем разделе демонстрируют механику переопределения методов, они не показывают его возможности. На самом деле, если бы для переопределения методов не существовало ничего, кроме соглашения о пространстве имен, то оно считалось бы в лучшем случае интересной особенностью, не обладая реальной ценностью. Тем не менее, это не так. Переопределение методов лежит в основе одной из самых мощных концепций Java — *диспетчеризации динамических методов*. Диспетчеризация динамических методов представляет собой механизм, с помощью которого вызов переопределенного метода распознается во время выполнения, а не на этапе компиляции. Динамическая диспетчеризация методов важна, потому что именно так в Java обеспечивается полиморфизм во время выполнения.

Давайте начнем с повторения важного принципа: ссылочная переменная типа суперкласса может ссылаться на объект подкласса. Данный факт используется в Java для распознавания вызовов переопределенных методов во время выполнения. А каким образом? Когда переопределенный метод вызывается через ссылку на суперкласс, версия метода, подлежащая выполнению, выясняется на основе типа объекта, на который производится ссылка в момент вызова. Соответственно, такое выяснение происходит во время выполнения. При ссылке на разные типы объектов будут вызываться разные версии переопределенного метода. Другими словами, *именно тип объекта, на который делается ссылка* (а не тип ссылочной переменной), определяет, какая версия переопределенного метода будет выполняться. Таким образом, если суперкласс содержит метод, который переопределяется в подклассе, то при ссылке на разные типы объектов через ссылочную переменную типа суперкласса выполняются разные версии метода.

Ниже показан пример, предназначенный для иллюстрации динамической диспетчеризации методов:

```
// Динамическая диспетчеризация методов.
class A {
  void callme() {
    System.out.println("Внутри метода callme() класса A");
  }
}

class B extends A {
  // Переопределить callme().
  void callme() {
    System.out.println("Внутри метода callme() класса B");
  }
}
class C extends A {
  // Переопределить callme().
  void callme() {
    System.out.println("Внутри метода callme() класса C");
  }
}
class Dispatch {
  public static void main(String[] args) {
    A a = new A();        // объект типа A
    B b = new B();        // объект типа B
    C c = new C();        // объект типа C

    A r;                  // получить ссылку типа A

    r = a;                // r ссылается на объект A
    r.callme();           // вызывается версия callme() из A

    r = b;                // r ссылается на объект B
    r.callme();           // вызывается версия callme() из B
    r = c;                // r ссылается на объект C
    r.callme();           // вызывается версия callme() из C
  }
}
```

Вот вывод, генерируемый программой:

```
Внутри метода callme() класса A
Внутри метода callme() класса B
Внутри метода callme() класса C
```

В программе создается один суперкласс по имени A и два его подкласса, B и C. Подклассы B и C переопределяют метод `callme()`, объявленный в A. Внутри метода `main()` объявляются объекты типов A, B и C. Кроме того, объявляется ссылка типа A по имени `r`. Затем в программе переменной `r` по очереди присваивается ссылка на каждый тип объекта и производится вызов метода `callme()`. В выводе видно, что выполняемая версия `callme()` определяется типом объекта, на который делается ссылка во время вызова. Если бы версия определялась типом ссылочной переменной `r`, то вы бы увидели три вызова метода `callme()` класса A.

На заметку! Читатели, знакомые с языком C++ или C#, найдут сходство переопределенных методов в Java с виртуальными функциями в этих языках.

Зачем нужны переопределенные методы?

Как было указано ранее, переопределенные методы позволяют Java поддерживать полиморфизм во время выполнения. Полиморфизм важен для ООП по одной причине: он позволяет универсальному классу определять методы, которые будут общими для всех производных от него классов, одновременно разрешая подклассам определять индивидуальные реализации некоторых или всех общих методов. Переопределенные методы — еще один способ, которым в Java обеспечивается аспект полиморфизма "один интерфейс, несколько методов".

Одним из ключей к успешному применению полиморфизма является понимание того, что суперклассы и подклассы образуют иерархию с продвижением от меньшей специализации к большей. При правильном использовании суперкласс предоставляет все элементы, которые подкласс может задействовать напрямую. Он также определяет те методы, которые производный класс должен реализовать самостоятельно. Это позволяет подклассу не только гибко определять собственные методы, но также обеспечивает согласованный интерфейс. Таким образом, комбинируя наследование с переопределенными методами, суперкласс может определять общую форму методов, которые будут потребляться всеми его подклассами.

Динамический полиморфизм во время выполнения — один из самых мощных механизмов, которыми ООП воздействует на многократное использование и надежность кода. Способность существующих библиотек кода вызывать методы для экземпляров новых классов без перекомпиляции с одновременным сохранением чистого абстрактного интерфейса является чрезвычайно мощным инструментом.

Применение переопределения методов

Давайте рассмотрим более реалистичный пример, в котором используется переопределение методов. В следующей программе создается суперкласс по имени `Figure` (фигура), который хранит размеры двумерного объекта. В нем также определен метод с именем `area()`, вычисляющий площадь объекта. От суперкласса `Figure` наследуются два подкласса — `Rectangle` (прямоугольник) и `Triangle` (треугольник). Каждый из этих подклассов переопределяет метод `area()`, так что он возвращает площадь соответственно прямоугольника и треугольника.

```
// Использование полиморфизма во время выполнения.
class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  double area() {
    System.out.println("Площадь для Figure не определена.");
    return 0;
  }
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // Переопределить area() для прямоугольника.
  double area() {
    System.out.println("Внутри area() для Rectangle.");
    return dim1 * dim2;
  }
}

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // Переопределить area() для прямоугольного треугольника.
  double area() {
    System.out.println("Внутри area() для Triangle.");
    return dim1 * dim2 / 2;
  }
}

class FindAreas {
  public static void main(String[] args) {
    Figure f = new Figure(10, 10);
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref;
```

```
    figref = r;
    System.out.println("Площадь равна " + figref.area());
    figref = t;
    System.out.println("Площадь равна " + figref.area());
    figref = f;
    System.out.println("Площадь равна " + figref.area());
  }
}
```

Ниже показан вывод программы:

```
Внутри area() для Rectangle.
Площадь равна 45
Внутри area() для Triangle.
Площадь равна 40
Площадь для Figure не определена.
Площадь равна 0
```

С помощью дуальных механизмов наследования и полиморфизма во время выполнения можно определить один согласованный интерфейс, который применяется несколькими разными, но связанными типами объектов. В данном случае, если объект является производным от `Figure`, то его площадь можно получить, вызвав метод `area()`. Интерфейс этой операции одинаков вне зависимости от того, какой тип фигуры используется.

Использование абстрактных классов

Бывают ситуации, когда желательно определить суперкласс, который объявляет структуру заданной абстракции, не предоставляя полные реализации методов. То есть иногда нужно создать суперкласс, определяющий только обобщенную форму, которая будет применяться всеми его подклассами, оставляя каждому подклассу возможность заполнить детали. Такой класс определяет природу методов, подлежащих реализации в подклассах. Ситуация подобного рода может возникнуть, когда суперкласс не способен создать осмысленную реализацию метода. Это относится к классу `Figure`, использованному в предыдущем примере. Определение метода `area()` является просто заполнителем. Он не будет рассчитывать и отображать площадь объекта любого вида.

При создании собственных библиотеки классов вы заметите, что нередко метод не имеет осмысленного определения в контексте своего суперкласса. Справиться с такой ситуацией можно двумя способами. Один из способов, как было показано в предыдущем примере, предусматривает просто выдачу предупреждающего сообщения. Хотя этот подход может быть полезен в определенных ситуациях, скажем, при отладке, обычно он не подходит. У вас могут быть методы, которые должны переопределяться в подклассе, чтобы подкласс имел какой-нибудь смысл. Возьмем класс `Triangle`. Он не имеет смысла, если метод `area()` не определен. В данном случае нужен какой-то способ гарантирования того, что подкласс действительно переопределяет все необходимые методы. В Java проблема решается посредством *абстрактных методов*.

Вы можете потребовать, чтобы некоторые методы были переопределены в подклассах, указав модификатор abstract. Иногда их называют методами, подпадающими под *ответственность подкласса*, потому что они не имеют реализации, указанной в суперклассе. Таким образом, подкласс обязан их переопределить — он не может просто использовать версию, определенную в суперклассе. Для объявления абстрактного метода применяется следующая общая форма:

```
abstract тип имя(список-параметров);
```

Как видите, тело метода отсутствует.

Любой класс, содержащий один или несколько абстрактных методов, тоже должен быть объявлен абстрактным. Чтобы объявить класс абстрактным, перед ключевым словом class в начале объявления класса просто используется ключевое слово abstract. Объектов абстрактного класса не бывает, т.е. экземпляр абстрактного класса нельзя создать напрямую с помощью операции new. Подобного рода объекты были бы бесполезными, т.к. абстрактный класс не определен полностью. Кроме того, не допускается объявлять абстрактные конструкторы или абстрактные статические методы. Любой подкласс абстрактного класса должен либо реализовать все абстрактные методы суперкласса, либо сам быть объявлен абстрактным.

Далее представлен простой пример класса с абстрактным методом, за которым следует класс, реализующий этот метод:

```
// Простая демонстрация применения abstract.
abstract class A {
  abstract void callme();

  // Конкретные методы в абстрактных классах по-прежнему разрешены.
  void callmetoo() {
    System.out.println("Это конкретный метод.");
  }
}
class B extends A {
  void callme() {
    System.out.println("Реализация callme() в классе B.");
  }
}
class AbstractDemo {
  public static void main(String[] args) {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```

Обратите внимание, что никаких объектов класса A в программе не объявлено. Как уже упоминалось, создать экземпляр абстрактного класса невозможно. Еще один момент: в классе A реализован конкретный метод callmetoo(), что вполне приемлемо. Абстрактные классы могут включать столько реализаций, сколько сочтут нужным.

Хотя абстрактные классы нельзя задействовать для создания объектов, их можно применять для создания ссылок на объекты, поскольку подход Java к полиморфизму во время выполнения обеспечивается через использование ссылок на суперклассы. Таким образом, должна быть возможность создания ссылки на абстрактный класс, чтобы ее можно было применять для указания на объект подкласса. Ниже вы увидите, как используется это средство.

Приведенный ранее класс `Figure` можно усовершенствовать за счет применения абстрактного класса. Поскольку для неопределенной двумерной фигуры нет осмысленного понятия площади, в следующей версии программы метод `area()` в `Figure` объявляется как абстрактный. Конечно, теперь все классы, производные от `Figure`, должны переопределять `area()`.

```
// Использование абстрактных методов и классов.
abstract class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  // Теперь area() - абстрактный метод.
  abstract double area();
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // Переопределить area() для прямоугольника.
  double area() {
    System.out.println("Внутри area() для Rectangle.");
    return dim1 * dim2;
  }
}

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // Переопределить area() для прямоугольного треугольника.
  double area() {
    System.out.println("Внутри area() для Triangle.");
    return dim1 * dim2 / 2;
  }
}

class AbstractAreas {
  public static void main(String[] args) {
    // Figure f = new Figure(10, 10);  // теперь недопустимо
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref;  // нормально, объект не создается
```

```
    figref = r;
    System.out.println("Площадь равна " + figref.area());

    figref = t;
    System.out.println("Площадь равна " + figref.area());
  }
}
```

Как видно из комментария внутри `main()`, объявлять объекты типа `Figure` больше невозможно, потому что теперь он абстрактный. И все подклассы `Figure` должны переопределять метод `area()`. Чтобы удостовериться в этом, попробуйте создать подкласс, не переопределяющий метод `area()`. Вы получите ошибку на этапе компиляции.

Несмотря на невозможность создания объекта типа `Figure`, разрешено создавать ссылочную переменную типа `Figure`. Переменная `figref` объявлена как ссылка на `Figure`, т.е. ее можно использовать для ссылки на объект любого класса, производного от `Figure`. Как объяснялось ранее, именно через ссылочные переменные типа суперкласса переопределенные методы распознаются во время выполнения.

Использование ключевого слова `final` с наследованием

Ключевое слово `final` применяется в трех ситуациях. Первая из них — его можно использовать для создания эквивалента именованной константы. Такое применение было описано в предыдущей главе. Две других ситуации использования `final` касаются наследования и рассматриваются далее.

Использование ключевого слова `final` для предотвращения переопределения

Наряду с тем, что переопределение методов является одной из самых мощных функциональных средств Java, иногда его желательно предотвращать. Чтобы запретить переопределение метода, в начале его объявления понадобится указать ключевое `final` в качестве модификатора. Методы, объявленные как `final`, не могут быть переопределены. Применение ключевого слова `final` демонстрируется в следующем фрагменте кода:

```
class A {
  final void meth() {
    System.out.println("Это метод final.");
  }
}

class B extends A {
  void meth() {  // ОШИБКА! Переопределять нельзя.
    System.out.println("Не разрешено!");
  }
}
```

Поскольку метод `meth()` объявлен как `final`, его нельзя переопределять в классе `B`. При попытке это сделать возникнет ошибка на этапе компиляции.

Методы, объявленные как `final`, иногда могут обеспечить повышение производительности: компилятор способен *встраивать* их вызовы, потому что он "знает", что они не будут переопределяться в подклассе. Когда компилятор Java встречает вызов небольшого метода `final`, он часто может копировать байт-код для подпрограммы непосредственно в скомпилированный код вызывающего метода, тем самым устраняя накладные расходы по вызову метода. Встраивание возможно только с методами `final`. Обычно компилятор Java распознает вызовы методов динамически во время выполнения. Это называется *поздним связыванием*. Но поскольку методы `final` не могут быть переопределены, их вызов может распознаваться на этапе компиляции. Это называется *ранним связыванием*.

Использование ключевого слова `final` для предотвращения наследования

Иногда нужно предотвратить наследование класса. Для этого перед объявлением класса укажите ключевое слово `final`. Объявление класса как `final` также неявно объявляет все его методы как `final`. Вполне ожидаемо объявлять класс как `abstract` и `final` одновременно не разрешено, поскольку абстрактный класс сам по себе неполный и в обеспечении полных реализаций полагается на свои подклассы.

Вот пример класса `final`:

```
final class A {
  // ...
}

// Следующий класс недопустим.
class B extends A {  // ОШИБКА! Создавать подкласс класса A нельзя
  // ...
}
```

В комментариях видно, что класс `B` не может быть унаследован от `A`, т.к. `A` объявлен с ключевым словом `final`.

На заметку! Начиная с версии JDK 17, в Java появилась возможность запечатывать класс. Запечатывание предлагает тонкий контроль над наследованием и рассматривается в главе 17.

Выведение типов локальных переменных и наследование

В главе 3 объяснялось, что в версии JDK 10 к языку Java было добавлено выведение типов локальных переменных, которое поддерживается контекстно-чувствительным ключевым словом `var`. Важно иметь четкое представление о том, как работает выведение типов в иерархии наследования. Вспомните, что ссылка на суперкласс может ссылаться на объект производного класса, и та-

кое средство является частью поддержки полиморфизма в Java. Однако важно помнить, что при использовании выведения типов локальных переменных выведенный тип переменной базируется на объявленном типе ее инициализатора. Следовательно, если инициализатор относится к типу суперкласса, то он и будет выведенным типом переменной. Не имеет значения, является ли фактический объект, на который ссылается инициализатор, экземпляром производного класса. Например, взгляните на показанную далее программу:

```
// При работе с наследованием выведенным типом является объявленный
// тип инициализатора и он может отличаться от производного
// типа объекта, на который ссылается инициализатор.
class MyClass {
  // ...
}
class FirstDerivedClass extends MyClass {
  int x;
  // ...
}
class SecondDerivedClass extends FirstDerivedClass {
  int y;
  // ...
}
class TypeInferenceAndInheritance {
  // Возвратить некоторый тип объекта MyClass.
  static MyClass getObj(int which) {
    switch(which) {
      case 0: return new MyClass();
      case 1: return new FirstDerivedClass();
      default: return new SecondDerivedClass();
    }
  }
  public static void main(String[] args) {
    // Несмотря на то что getObj() возвращает различные типы
    // объектов в иерархии наследования MyClass, объявленным
    // типом возвращаемого значения является MyClass.
    // В результате во всех трех показанных здесь случаях
    // предполагается, что типом переменных является MyClass,
    // хотя получаются разные производные типы объектов.

    // В этом случае getObj() возвращает объект MyClass.
    var mc = getObj(0);

    // В этом случае getObj() возвращает объект FirstDerivedClass.
    var mc2 = getObj(1);

    // В этом случае getObj() возвращает объект SecondDerivedClass.
    var mc3 = getObj(2);

    // Поскольку типы mc2 и mc3 выводятся как MyClass (т.к. возвращаемым типом
    // getObj() является MyClass), то ни mc2, ни mc3 не могут получить доступ
    // к полям, объявленным в FirstDerivedClass или SecondDerivedClass.
    // mc2.x = 10;  // Ошибка! Класс MyClass не имеет поля x.
    // mc3.y = 10;  // Ошибка' Класс MyClass не имеет поля y.
  }
}
```

В программе создается иерархия, состоящая из трех классов, на вершине которых находится MyClass. Класс FirstDerivedClass определен как подкласс MyClass, а SecondDerivedClass — как подкласс FirstDerivedClass. Затем с применением выведения типа создаются три переменные с именами mc, mc2 и mc3 путем вызова getObj(). Метод getObj() имеет возвращаемый тип MyClass (суперкласс), но в зависимости от передаваемого аргумента возвращает объекты типа MyClass, FirstDerivedClass или SecondDerivedClass. Как видно в отображенных результатах программы, выведенный тип определяется возвращаемым типом getObj(), а не фактическим типом полученного объекта. Таким образом, все три переменные будут иметь тип MyClass.

Класс Object

Существует один особый класс Object, определенный в Java. Все остальные классы являются подклассами Object, т.е. Object представляет собой суперкласс для всех остальных классов. Это означает, что ссылочная переменная типа Object может ссылаться на объект любого другого класса. Кроме того, поскольку массивы реализованы в виде классов, переменная типа Object также может ссылаться на любой массив.

В классе Object определены методы, описанные в табл. 8.1, которые доступны в любом объекте.

Таблица 8.1. Методы класса Object

Метод	Назначение
`Object clone()`	Создает новый объект, который совпадает с клонируемым объектом
`boolean equals(Object` *объект*`)`	Определяет, равен ли один объект другому
`void finalize()`	Вызывается перед удалением неиспользуемого объекта. (Объявлен устаревшим в JDK 9.)
`Class<?> getClass()`	Получает класс объекта во время выполнения
`int hashCode()`	Возвращает хеш-код, ассоциированный с вызывающим объектом
`void notify()`	Возобновляет выполнение потока, ожидающего вызывающий объект
`void notifyAll()`	Возобновляет выполнение всех потоков, ожидающих вызывающий объект
`String toString()`	Возвращает строку, которая описывает объект

Окончание табл. 8.1

Метод	Назначение
`void wait()`	Ожидает другого потока выполнения
`void wait(` `  long` *миллисекунд*`)`	

`void wait(`
`  long` *миллисекунд*`,`
`  int` *наносекунд*`)`	

Методы `getClass()`, `notify()`, `notifyAll()` и `wait()` объявлены как `final`. Остальные методы можно переопределять. Перечисленные в табл. 8.1 методы описаны в других местах книги. Тем не менее, обратите сейчас внимание на два метода: `equals()` и `toString()`. Метод `equals()` сравнивает два объекта. Он возвращает `true`, если объекты равны, и `false` в противном случае. Точное определение равенства может варьироваться в зависимости от типа сравниваемых объектов. Метод `toString()` возвращает строку, содержащую описание объекта, для которого он вызывается. Также этот метод вызывается автоматически, когда объект выводится с помощью `println()`. Многие классы переопределяют метод `toString()`, что позволяет им адаптировать описание специально для типов объектов, которые они создают.

И последнее замечание: обратите внимание на необычный синтаксис возвращаемого типа для `getClass()`. Он имеет отношение к средству *обобщений* языка Java, которое будет рассматриваться в главе 14.

ГЛАВА

9 Пакеты и интерфейсы

В этой главе рассматриваются два самых инновационных средства Java: пакеты и интерфейсы. *Пакеты* представляют собой контейнеры для классов. Они используются для отделения пространства имен класса. Например, создав класс по имени `List` и сохранив его в собственном пакете, можно не беспокоиться о том, что он будет конфликтовать с другим классом по имени `List`, который находится где-то в другом месте. Пакеты хранятся в иерархическом порядке и явно импортируются в определения новых классов. Как будет показано в главе 16, пакеты также играют важную роль в модулях.

В предыдущих главах вы видели, что методы определяют интерфейс к данным в классе. Ключевое слово `interface` позволяет полностью абстрагировать интерфейс от его реализации. С помощью `interface` указывается набор методов, которые могут быть реализованы одним или несколькими классами. В своей традиционной форме интерфейс сам по себе не определяет никакой реализации. Хотя интерфейсы похожи на абстрактные классы, они обладают дополнительной возможностью: класс может реализовывать более одного интерфейса. В противоположность этому класс может быть унаследован только от одного суперкласса (абстрактного либо иного).

Пакеты

В предыдущих главах имя каждого примера класса принадлежало одному и тому же пространству имен. Таким образом, каждому классу необходимо было назначать уникальное имя, чтобы избежать конфликтов имен. Без какого-либо способа управления пространством имен через некоторое время удобные описательные имена для отдельных классов могут попросту закончиться. Кроме того, нужно каким-то образом гарантировать то, что имя, выбранное для класса, будет достаточно уникальным и не конфликтующим с именами, которые другие программисты назначили своим классам. (Представьте себе небольшую группу программистов, спорящих за право использовать "Foobar" в качестве имени класса. Или вообразите ситуацию, когда все сообщество в Интернете выясняет, кто первым назвал класс "Espresso".) К счастью, в Java предоставляется механизм для разделения пространства имен классов на более управляемые фрагменты — пакеты. Пакет является

как механизмом именования, так и механизмом управления видимостью. Вы можете определять классы внутри пакета, которые не доступны коду вне пакета. Вы также можете определять члены класса, которые видны только другим членам классов в том же пакете. Это позволяет вашим классам хорошо знать друг друга, но не раскрывать такие знания остальному миру.

Определение пакета

Создать пакет довольно легко: понадобится просто поместить в начало файла с исходным кодом Java оператор `package`. Любые классы, объявленные в данном файле, будут принадлежать указанному пакету. Оператор `package` определяет пространство имен, в котором хранятся классы. Если оператор `package` отсутствует, тогда имена классов помещаются в стандартный пакет, не имеющий имени. (Вот почему раньше вам не приходилось беспокоиться о пакетах.) Хотя стандартный пакет пригоден в коротких учебных программах, он не подходит для реальных приложений. Вы будете определять пакет для своего кода почти всегда. Вот общая форма оператора `package`:

```
package пакет;
```

Здесь `пакет` представляет имя пакета. Например, следующий оператор создает пакет по имени `mypackage`:

```
package mypackage;
```

Как правило, в Java для хранения пакетов применяются каталоги файловой системы, и именно такой подход задействован в примерах, приводимых в книге. Скажем, файлы `.class` для любых классов, которые объявляются как часть `mypackage`, должны храниться в каталоге с именем `mypackage`. Помните о том, что регистр символов имеет значение, а имя каталога должно точно совпадать с именем пакета.

Один и тот же оператор `package` может находиться в нескольких файлах. Оператор `package` лишь указывает, к какому пакету принадлежат классы, определенные в файле. Это не исключает, что другие классы в других файлах могут быть частью того же самого пакета. Большинство пакетов в реальных приложениях разнесено по многим файлам.

Допускается создавать иерархию пакетов, для чего нужно просто отделять имя каждого пакета от имени пакета над ним с помощью точки. Общая форма оператора многоуровневого пакета выглядит следующим образом:

```
package пакет1[.пакет2[.пакет3]];
```

Иерархия пакетов должна быть отражена в файловой системе на машине для разработки приложений Java. Например, объявленный ниже пакет должен храниться в папке `a\b\c` в среде Windows:

```
package a.b.c;
```

Имена пакетов должны выбираться крайне аккуратно, т.к. нельзя переименовать пакет, не переименовав каталог, в котором хранятся классы.

Поиск пакетов и CLASSPATH

Как только что объяснялось, пакеты обычно отражаются посредством каталогов. Тогда возникает важный вопрос: как исполняющая среда Java узнает, где искать создаваемые вами пакеты? Что касается примеров в этой главе, то ответ состоит из трех частей. Во-первых, по умолчанию исполняющая среда Java в качестве начальной точки использует текущий рабочий каталог. Таким образом, если ваш пакет расположен в каком-то подкаталоге внутри текущего каталога, то он будет найден. Во-вторых, вы можете указать путь или пути к каталогам, установив переменную среды CLASSPATH. В-третьих, вы можете применить параметр `-classpath` при запуске `java` и `javac`, чтобы указать путь к своим классам. Полезно отметить, что начиная с JDK 9, пакет может быть частью модуля и потому находиться в пути к модулю. Однако обсуждение модулей и путей к модулям откладывается до главы 16. Сейчас мы будем использовать только пути к классам.

Например, рассмотрим следующую спецификацию пакета:

```
package mypack;
```

Чтобы программа могла найти пакет `mypack`, ее можно либо запустить из каталога непосредственно над `mypack`, либо переменная среды CLASSPATH должна включать путь к `mypack`, либо при запуске программы через `java` в параметре `-classpath` должен быть указан путь к `mypack`.

Когда применяются последние два способа, путь к классу *не должен* содержать само имя `mypack`. Он должен просто указывать *и путь к* `mypack`. Скажем, если в среде Windows путем к `mypack` является:

```
C:\MyPrograms\Java\mypack
```

тогда путем к классу для `mypack` будет:

```
C:\MyPrograms\Java
```

Испытать примеры, приведенные в книге, проще всего, создав каталоги пакетов внутри текущего каталога разработки, поместив файлы `.class` в соответствующие каталоги и затем запустив программы из каталога разработки. Именно такой подход используется в рассматриваемом далее примере.

Краткий пример пакета

Приняв к сведению предыдущее обсуждение, можете испытать показанный ниже простой пакет:

```
// Простой пакет.
package mypack;

class Balance {
  String name;
  double bal;

  Balance(String n, double b) {
    name = n;
    bal = b;
  }
```

```
  void show() {
    if(bal<0)
    System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}

class AccountBalance {
  public static void main(String[] args) {
    Balance[] current = new Balance[3];

    current[0] = new Balance("K. J. Fielding", 123.23);
    current[1] = new Balance("Will Tell", 157.02);
    current[2] = new Balance("Tom Jackson", -12.33);

    for(int i=0; i<3; i++) current[i].show();
  }
}
```

Назначьте файлу имя `AccountBalance.java` и поместите его в каталог `mypack`.

Скомпилируйте файл `AccountBalance.java`. Удостоверьтесь в том, что результирующий файл `.class` тоже находится в каталоге `mypack`. Затем попробуйте выполнить класс `AccountBalance` с применением следующей команды:

```
java mypack.AccountBalance
```

Помните, что при вводе этой команды вы должны находиться в каталоге выше `mypack`. (Кроме того, для указания пути к `mypack` вы можете прибегнуть к одному из двух других способов, описанных в предыдущем разделе.)

Как объяснялось ранее, класс `AccountBalance` теперь является частью пакета `mypack`, т.е. он не может быть выполнен сам по себе. Другими словами, вы не можете использовать такую команду:

```
java AccountBalance
```

Класс `AccountBalance` должен быть уточнен именем пакета.

Пакеты и доступ к членам классов

В предшествующих главах вы ознакомились с различными аспектами механизма управления доступом в Java и его модификаторами доступа. Например, вы уже знаете, что доступ к закрытому члену класса предоставляется только другим членам этого класса. Пакеты добавляют еще одно измерение к управлению доступом. Вы увидите, что Java предоставляет множество уровней защиты, которые позволяют с высокой степенью детализации управлять видимостью переменных и методов внутри классов, подклассов и пакетов.

Классы и пакеты являются средствами инкапсуляции и содержания в себе пространства имен, а также области видимости переменных и методов. Пакеты действуют в качестве контейнеров для классов и других подчиненных пакетов. Классы действуют как контейнеры для данных и кода. Класс — это наименьшая единица абстракции Java.

Что касается взаимодействия между классами и пакетами Java, то существуют четыре категории видимости для членов класса:

- подклассы в том же самом пакете;
- не подклассы в том же самом пакете;
- подклассы в других пакетах;
- классы, которые не находятся в том же самом пакете и не являются подклассами.

Три модификатора доступа, `private`, `public` и `protected`, предоставляют различные способы создания множества уровней доступа, требуемых этими категориями. Взаимосвязь между ними описана в табл. 9.1.

Таблица 9.1. Доступ к членам классов

	private	Без модификатора	protected	public
Тот же класс	Да	Да	Да	Да
Подкласс из того же пакета	Нет	Да	Да	Да
Не подкласс из того же пакета	Нет	Да	Да	Да
Подкласс из другого пакета	Нет	Нет	Да	Да
Не подкласс из другого пакета	Нет	Нет	Нет	Да

Хотя механизм управления доступом в Java может показаться сложным, его можно упростить следующим образом. Ко всему, что объявлено как `public`, можно получать доступ из разных классов и разных пакетов. Все, что объявлено как `private`, не может быть видимым за пределами его класса. Когда у члена нет явной спецификации доступа, он виден подклассам, а также другим классам в том же пакете. Такой доступ принят по умолчанию. Если вы хотите, чтобы элемент был видимым за пределами вашего текущего пакета, но только классам, которые напрямую являются подклассами вашего класса, тогда объявите этот элемент как `protected`.

Правила доступа к членам классов, приведенные в табл. 9.1, применимы только к членам классов. Класс, не являющийся вложенным, имеет только два возможных уровня доступа: стандартный и открытый. Когда класс объявлен как `public`, он доступен за пределами своего пакета. Если класс имеет стандартный доступ, то к нему может получать доступ только другой код в том же пакете. Когда класс является открытым, он должен быть единственным открытым классом, объявленным в файле, а файл должен иметь такое же имя, как у класса.

На заметку! Средство модулей также может влиять на доступность. Модули описаны в главе 16.

Пример, демонстрирующий использование модификаторов доступа

В показанном далее примере демонстрируются все комбинации модификаторов управления доступом. В примере имеются два пакета и пять классов. Не забывайте, что классы для двух разных пакетов должны храниться в каталогах с именами, которые совпадают с именами соответствующих пакетов — в данном случае p1 и p2.

В файле с исходным кодом для первого пакета определены три класса: Protection, Derived и SamePackage. В первом классе определяются четыре переменных типа int с каждым допустимым режимом доступа. Переменная n объявлена со стандартным доступом, n_pri — с доступом private, n_pro — с доступом protected, а n_pub — с доступом public.

Все последующие классы в рассматриваемом примере будут пытаться получить доступ к переменным экземпляра класса Protection. Строки, которые не будут компилироваться из-за ограничений доступа, закомментированы. Перед каждой из таких строк находится комментарий с перечислением мест, из которых этот уровень защиты разрешит доступ.

Второй класс, Derived, является подклассом Protection в том же пакете p1, что дает Derived доступ ко всем переменным в Protection кроме закрытой переменной n_pri. Третий класс, SamePackage, не является подклассом Protection, но находится в том же самом пакете и тоже имеет доступ ко всем переменным кроме n_pri.

Вот содержимое файла Protection.java:

```
package p1;

public class Protection {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;

  public Protection() {
    System.out.println("Конструктор базового класса");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

Ниже приведено содержимое файла Derived.java:

```
package p1;

class Derived extends Protection {
  Derived() {
    System.out.println("Конструктор производного класса");
    System.out.println("n = " + n);
```

```
    // Только класс.
    // System.out.println("n_pri = "4 + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

Далее показано содержимое файла SamePackage.java:

```
package p1;
class SamePackage {
  SamePackage() {
    Protection p = new Protection();
    System.out.println("Конструктор класса из того же пакета");
    System.out.println("n = " + p.n);
    // Только класс.
    // System.out.println("n_pri = " + p.n_pri);
    System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

Ниже представлен исходный код другого пакета, p2. Два класса, определенные в p2, охватывают остальные два условия, на которые влияет управление доступом. Первый класс, Protection2, является подклассом p1.Protection, что дает ему доступ ко всем переменным p1.Protection кроме n_pri (поскольку она закрытая) и n — переменной, объявленной со стандартной защитой.

Вспомните, что по умолчанию разрешен доступ только из класса или пакета, а не из подклассов вне пакета. Наконец, класс OtherPackage имеет доступ только к одной переменной n_pub, которая была объявлена открытой.

Вот содержимое файла Protection2.java:

```
package p2;
class Protection2 extends p1.Protection {
  Protection2() {
    System.out.println("Конструктор производного класса из другого пакета");
    // Только класс или пакет.
    // System.out.println("n = " + n);
    // Только класс.
    // System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

Далее приведено содержимое OtherPackage.java:

```
package p2;
class OtherPackage {
```

```
  OtherPackage() {
    p1.Protection p = new p1.Protection();
    System.out.println("Конструктор класса из другого пакета");
    // Только класс или пакет.
    // System.out.println("n = " + p.n);
    // Только класс.
    // System.out.println("n_pri = " + p.n_pri);
    // Только класс, подкласс или пакет.
    // System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

Если вы хотите испытать два созданных пакета, то ниже предлагаются два тестовых файла, которыми можно воспользоваться. Вот тестовый файл для пакета p1:

```
// Тестирование пакета p1.
package p1;
// Создать экземпляры различных классов в p1.
public class Demo {
  public static void main(String[] args) {
    Protection ob1 = new Protection();
    Derived ob2 = new Derived();
    SamePackage ob3 = new SamePackage();
  }
}
```

А вот тестовый файл для пакета p2:

```
// Тестирование пакета p2.
package p2;
// Создать экземпляры различных классов в p2.
public class Demo {
  public static void main(String[] args) {
    Protection2 ob1 = new Protection2();
    OtherPackage ob2 = new OtherPackage();
  }
}
```

Импортирование пакетов

С учетом того, что пакеты существуют и являются хорошим механизмом отделения различных классов друг от друга, легко понять, почему все встроенные классы Java хранятся в пакетах. В стандартном пакете без имени нет ни одного базового класса Java; все стандартные классы хранятся в каком-то именованном пакете. Поскольку классы в пакетах должны полностью уточняться с помощью одного или нескольких имен пакетов, набор длинного пути к пакету, разделенного точками, для каждого используемого класса может стать утомительным. По этой причине в составе Java имеется оператор импортирования `import`, который позволяет сделать видимыми определенные классы или целые пакеты. После импортирования на класс можно ссылаться

напрямую с применением только его имени. Оператор import является удобным инструментом для программиста и формально не нужен для написания законченной программы на Java. Тем не менее, если вы собираетесь ссылаться на несколько десятков классов в своем приложении, то оператор import существенно сократит объем набора.

В файле с исходным кодом на Java операторы import располагаются сразу после оператора package (если он есть) и перед любыми определениями классов. Общая форма оператора import выглядит следующим образом:

```
import пакет1[.пакет2].(имя-класса | *);
```

Здесь pkg1 — имя пакета верхнего уровня, а pkg2 — имя подчиненного пакета внутри внешнего пакета, отделенное точкой. На практике ограничения на глубину иерархии пакетов отсутствуют за исключением тех, что накладываются файловой системой. В конце оператора import указывается либо явное имя класса (*имя-класса*), либо звездочка (*), которая сообщает компилятору Java о необходимости импортирования всего пакета. Ниже демонстрируется использование обеих форм:

```
import java.util.Date;
import java.io.*;
```

Все стандартные классы Java SE, входящие в состав Java, начинаются с имени java. Базовые языковые функции хранятся в пакете по имени java.lang. Обычно вам приходится импортировать каждый пакет или класс, с которым планируется работа, но поскольку язык Java бесполезен без значительной части функциональности пакета java.lang, он неявно импортируется компилятором для всех программ. Это эквивалентно наличию в начале кода всех ваших программ следующей строки:

```
import java.lang.*;
```

Если в двух разных пакетах, импортированных с помощью оператора import со звездочкой, существуют классы с одинаковыми именами, то компилятор никак не будет реагировать, пока вы не попытаетесь использовать один из классов. В таком случае возникнет ошибка на этапе компиляции и нужно будет вместе с именем класса явно указать пакет.

Важно подчеркнуть, что оператор import необязателен. В любом месте, где указывается имя класса, можно применять его *полностью уточненное имя*, которое включает в себя всю иерархию пакетов. Например, в следующем фрагменте кода используется оператор import:

```
import java.util.*;
class MyDate extends Date {
}
```

А вот как можно переписать фрагмент без оператора import:

```
class MyDate extends java.util.Date {
}
```

В этой версии класс Date указан с применением полностью уточненного имени.

Как было показано в табл. 9.1, при импортировании пакета классам внутри импортирующего кода, которые не являются подклассами, будут доступны только элементы из пакета, объявленные как public. Например, если вы хотите, чтобы класс Balance из приведенного ранее пакета mypack был доступен как автономный класс для общего использования за пределами mypack, то вам понадобится объявить его как public и поместить в отдельный файл:

```
package mypack;

/* Теперь класс Balance, его конструктор и метод show() стали открытыми.
   Это означает, что они могут использоваться в коде классов,
   не являющихся подклассами, вне их пакета.
*/
public class Balance {
  String name;
  double bal;

  public Balance(String n, double b) {
    name = n;
    bal = b;
  }

  public void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}
```

Теперь класс Balance стал открытым, равно как его конструктор и метод show(). Это означает, что к ним можно получить доступ в коде любого вида за рамками пакета mypack. Например, класс TestBalance импортирует mypack и затем может работать с классом Balance:

```
import mypack.*;
class TestBalance {
  public static void main(String[] args) {

    /* Поскольку Balance открыт, вы можете использовать
       класс Balance и вызывать его конструктор. */
    Balance test = new Balance("J. J. Jaspers", 99.88);

    test.show();  // вы можете также вызывать метод show()
  }
}
```

В качестве эксперимента удалите модификатор доступа public из класса Balance и попробуйте скомпилировать TestBalance. Как объяснялось ранее, возникнут ошибки на этапе компиляции.

Интерфейсы

С помощью ключевого слова interface вы можете полностью абстрагировать интерфейс класса от его реализации. То есть с применением interface можно указать, что класс должен делать, но не как конкретно. Интерфейсы

синтаксически похожи на классы, но в них отсутствуют переменные экземпляра и, как правило, их методы объявляются без тела. На практике это означает, что вы можете определять интерфейсы, не делая предположений о том, каким образом они реализованы. После определения интерфейс может быть реализован любым количеством классов. Кроме того, один класс может реализовывать любое количество интерфейсов.

Для реализации интерфейса класс должен предоставить полный набор методов, требуемых интерфейсом. Однако каждый класс может самостоятельно определять детали собственной реализации. За счет предоставления ключевого слова `interface` язык Java позволяет в полной мере задействовать аспект полиморфизма "один интерфейс, несколько методов".

Интерфейсы предназначены для поддержки динамического распознавания методов во время выполнения. Обычно для вызова метода из одного класса внутри другого оба класса должны присутствовать во время компиляции, чтобы компилятор Java мог проверить совместимость сигнатур методов. Такое требование само по себе создает статическую и нерасширяемую среду создания классов. В системе подобного рода функциональность неизбежно поднимается все выше и выше в иерархии классов, так что механизмы становятся доступными для все большего числа подклассов. Интерфейсы предназначены для того, чтобы решить эту проблему. Они отделяют определение метода или набора методов от иерархии наследования. Поскольку интерфейсы находятся в иерархии, отличающейся от иерархии классов, у классов, не связанных с точки зрения иерархии классов, появляется возможность реализации одного и того же интерфейса. Именно здесь проявляется настоящая сила интерфейсов.

Определение интерфейса

Интерфейс определяется во многом подобно классу. Ниже показана упрощенная общая форма интерфейса:

```
доступ interface имя {
  возвращаемый-тип имя-метода1(список-параметров);
  возвращаемый-тип имя-метода2(список-параметров);

  тип финальное-имя-переменной1 = значение;
  тип финальное-имя-переменной2 = значение;
  // ...
  возвращаемый-тип имя-методаN(список-параметров);
  тип финальное-имя-переменнойN = значение;
}
```

Если модификатор доступа отсутствует, тогда устанавливается стандартный доступ и интерфейс будет доступным только другим элементам пакета, в котором он объявлен. Когда интерфейс объявлен как `public`, его может использовать код вне пакета, где он объявлен. В таком случае интерфейс должен быть единственным открытым интерфейсом, объявленным в файле, а файл должен иметь то же имя, что и интерфейс. Имя интерфейса может быть

любым допустимым идентификатором. Обратите внимание, что объявленные методы не имеют тела. Они заканчиваются точкой с запятой после списка параметров. По существу это абстрактные методы. Каждый класс, включающий такой интерфейс, обязан реализовывать все методы.

Прежде чем продолжить, необходимо сделать важное замечание. В версии JDK 8 к интерфейсу было добавлено функциональное средство, значительно изменившее его возможности. До JDK 8 в интерфейсе нельзя было определять какие-то реализации. Речь идет о виде интерфейса, упрощенная форма которого представлена выше, где ни одно объявление метода не снабжалось телом. Таким образом, до выхода JDK 8 интерфейс мог определять только "что", но не "как". В версии JDK 8 ситуация изменилась. Начиная с JDK 8, к методу интерфейса можно добавлять *стандартную реализацию*. Кроме того, в JDK 8 также добавлены статические методы интерфейса, а начиная с JDK 9, интерфейс может включать закрытые методы. В результате теперь интерфейс может задавать какое-то поведение. Тем не менее, такие методы представляют собой то, что по существу является средствами специального назначения, и первоначальный замысел интерфейса по-прежнему остается. Поэтому, как правило, вы все еще будете часто создавать и использовать интерфейсы, в которых новые средства не применяются. По указанной причине мы начнем с обсуждения интерфейса в его традиционной форме. Более новые средства интерфейса описаны в конце главы.

Как показывает общая форма, внутри объявлений интерфейса можно объявлять переменные. Они неявно являются `final` и `static`, т.е. не могут изменяться реализующим классом. Они также должны быть инициализированы. Все методы и переменные неявно открыты.

Ниже приведен пример определения интерфейса, в котором объявляется простой интерфейс с единственным методом `callback()`, принимающим один целочисленный параметр:

```
interface Callback {
  void callback(int param);
}
```

Реализация интерфейсов

После определения интерфейса один или несколько классов могут его реализовать. Для реализации интерфейса включите в определение класса конструкцию `implements` и затем создайте методы, требуемые интерфейсом. Общая форма класса, содержащего конструкцию `implements`, выглядит следующим образом:

```
class имя-класса [extends суперкласс] [implements интерфейс [, интерфейс ...]] {
  // тело класса
}
```

Если класс реализует более одного интерфейса, тогда интерфейсы отделяются друг от друга запятыми. Если класс реализует два интерфейса, в которых объявлен один и тот же метод, то тот же самый метод будет использоваться

клиентами любого из двух интерфейсов. Методы, реализующие интерфейс, должны быть объявлены как public. Кроме того, сигнатура типов реализующего метода должна в точности совпадать с сигнатурой типов, указанной в определении интерфейса.

Далее показан небольшой пример класса, который реализует определенный ранее интерфейс Callback:

```
class Client implements Callback {
  // Реализовать метод интерфейса Callback.
  public void callback(int p) {
    System.out.println("callback() вызывается с " + p);
  }
}
```

Обратите внимание, что метод callback() объявлен с применением модификатора доступа public.

Помните! При реализации метода интерфейса он должен быть объявлен как public.

Для классов, реализующих интерфейсы, допустимо и распространено определение собственных дополнительных членов. Например, в следующей версии класса Client реализуется callback() и добавляется метод nonIfaceMeth():

```
class Client implements Callback {
  // Реализовать метод интерфейса Callback.
  public void callback(int p) {
    System.out.println("callback() вызывается со значением " + p);
  }

  void nonIfaceMeth() {
    System.out.println("Классы, которые реализуют интерфейсы, " +
                       "могут также определять и другие члены.");
  }
}
```

Доступ к реализациям через ссылки на интерфейсы

Вы можете объявлять переменные как ссылки на объекты, в которых применяется тип интерфейса, а не тип класса. С помощью переменной подобного рода можно ссылаться на любой экземпляр любого класса, реализующего объявленный интерфейс. При вызове метода через одну из таких ссылок корректная версия будет вызываться на основе фактического экземпляра реализации интерфейса, на который осуществляется ссылка. Это одна из ключевых особенностей интерфейсов. Поиск метода, подлежащего выполнению, производится динамически во время выполнения, что позволяет создавать классы позже, чем код, вызывающий их методы. Диспетчеризация методов в вызывающем коде возможна через интерфейс, не требуя каких-либо знаний о "вызываемой стороне". Такой процесс аналогичен использованию ссылки на суперкласс для доступа к объекту подкласса, как было описано в главе 8.

В следующем примере метод `callback()` вызывается через переменную ссылки на интерфейс:

```
class TestIface {
  public static void main(String[] args) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

Вот вывод, генерируемый программой:

```
callback() вызывается со значением 42
```

Обратите внимание, что переменная `c` объявлена с типом интерфейса `Callback`, хотя ей был присвоен экземпляр класса `Client`. Хотя переменную `c` можно использовать для доступа к методу `callback()`, через нее не удастся получить доступ ни к каким другим членам класса `Client`. Переменной ссылки на интерфейс известны только методы, присутствующие в ее объявлении `interface`. Таким образом, переменную `c` нельзя применять для доступа к методу `nonIfaceMeth()`, поскольку он определен в `Client`, а не в `Callback`.

Хотя в предыдущем примере было показано, каким образом переменная ссылки на интерфейс может получить доступ к объекту реализации, в нем не демонстрировалась полиморфные возможности такой ссылки. Чтобы испытать их, сначала понадобится создать еще одну реализацию `Callback`:

```
// Еще одна реализация Callback.
class AnotherClient implements Callback {
  // Реализовать метод интерфейса Callback.
  public void callback(int p) {
    System.out.println("Еще одна версия callback()");
    System.out.println("p в квадрате равно " + (p*p));
  }
}
```

Теперь создадим следующий класс:

```
class TestIface2 {
  public static void main(String[] args) {
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();
    c.callback(42);
    c = ob;                   // c теперь ссылается на объект AnotherClient
    c.callback(42);
  }
}
```

Ниже показан вывод, генерируемый программой:

```
callback() вызывается со значением 42
Еще одна версия callback()
p в квадрате равно 1764
```

Как видите, вызываемая версия `callback()` определяется типом объекта, на который переменная `c` ссылается во время выполнения. Вскоре вы увидите другой, более реальный пример.

Частичные реализации

Если класс включает интерфейс, но не полностью реализует методы, требуемые этим интерфейсом, то такой класс должен быть объявлен абстрактным, например:

```
abstract class Incomplete implements Callback {
  int a, b;

  void show() {
    System.out.println(a + " " + b);
  }
  // ...
}
```

Здесь класс `Incomplete` не реализует метод `callback()` и должен быть объявлен абстрактным. Любой класс, который наследует `Incomplete`, обязан реализовывать `callback()` или сам должен быть объявлен как `abstract`.

Вложенные интерфейсы

Интерфейс может быть объявлен членом класса или другого интерфейса. Такой интерфейс называется *членом-интерфейсом* или *вложенным интерфейсом*. Вложенный интерфейс может быть объявлен как `public`, `private` или `protected`. Он отличается от интерфейса верхнего уровня, который должен быть либо объявлен как `public`, либо использовать стандартный уровень доступа, как было описано ранее. Когда вложенный интерфейс применяется за пределами своей области видимости, то он должен быть уточнен именем класса или интерфейса, членом которого является. Таким образом, вне класса или интерфейса, где объявлен вложенный интерфейс, его имя должно быть полностью уточненным.

Вот пример, в котором демонстрируется вложенный интерфейс:

```
// Пример вложенного интерфейса.

// Класс A содержит член-интерфейс.
class A {
  // Вложенный интерфейс.
  public interface NestedIF {
    boolean isNotNegative(int x);
  }
}

// Класс B реализует вложенный интерфейс.
class B implements A.NestedIF {
  public boolean isNotNegative(int x) {
    return x < 0 ? false: true;
  }
}
class NestedIFDemo {
  public static void main(String[] args) {

    // Использовать ссылку на вложенный интерфейс.
    A.NestedIF nif = new B();
```

```
    if(nif.isNotNegative(10))
      System.out.println("10 не является отрицательным");
    if(nif.isNotNegative(-12))
      System.out.println("Это выводиться не будет");
  }
}
```

Обратите внимание, что в классе A определен и объявлен открытым член-интерфейс по имени NestedIF. Затем вложенный интерфейс реализуется в классе B за счет указания следующей конструкции:

```
implements A.NestedIF
```

Кроме того, имя полностью уточнено именем объемлющего класса. Внутри метода main() создается ссылка A.NestedIF по имени nif, которой присваивается ссылка на объект B. Поскольку B реализует A.NestedIF, такая операция допустима.

Применение интерфейсов

Чтобы оценить возможности интерфейсов, мы обратимся к более реалистичному примеру. В предшествующих главах был разработан класс Stack, реализующий простой стек фиксированного размера. Однако реализовать стек можно многими способами. Например, стек может иметь фиксированный размер или быть "расширяемым". Стек также может храниться в виде массива, связного списка, двоичного дерева и т.д. Как бы ни был реализован стек, интерфейс со стеком остается неизменным, т.е. методы push() и pop() определяют интерфейс к стеку независимо от деталей реализации. Поскольку интерфейс к стеку отделен от его реализации, легко определить интерфейс стека, предоставив каждой реализации возможность определять специфику. Давайте рассмотрим два примера.

Для начала вот определение интерфейса для целочисленного стека. Поместите код в файл с именем IntStack.java. Данный интерфейс будет использоваться обеими реализациями стека.

```
// Определить интерфейс для целочисленного стека.
interface IntStack {
  void push(int item);        // сохранить элемент
  int pop();                  // извлечь элемент
}
```

В следующей программе создается класс по имени FixedStack, который реализует версию целочисленного стека с фиксированной длиной:

```
// Реализация IntStack, использующая хранилище фиксированной длины.
class FixedStack implements IntStack {
  private int[] stck;
  private int tos;
  // Разместить в памяти и инициализировать стек.
  FixedStack(int size) {
    stck = new int[size];
    tos = -1;
  }
```

```
  // Поместить элемент в стек.
  public void push(int item) {
    if(tos==stck.length-1)  // использовать член length
      System.out.println("Стек полон.");
    else
      stck[++tos] = item;
  }

  // Извлечь элемент из стека.
  public int pop() {
    if(tos < 0) {
      System.out.println("Стек опустошен.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class IFTest {
  public static void main(String[] args) {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);

    // Поместить несколько чисел в стеки.
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // Извлечь эти числа из стеков.
    System.out.println("Стек в mystack1:");
    for(int i=0; i<5; i++)
      System.out.println(mystack1.pop());

    System.out.println("Стек в mystack2:");
    for(int i=0; i<8; i++)
      System.out.println(mystack2.pop());
  }
}
```

Ниже приведена еще одна реализация `IntStack`, которая создает динамический стек с применением того же определения интерфейса. В этой реализации каждый стек создается с начальной длиной. В случае превышения начальной длины размер стека увеличивается. Каждый раз, когда требуется больше места, размер стека удваивается.

```
// Реализовать "расширяемый" стек.
class DynStack implements IntStack {
  private int[] stck;
  private int tos;

  // Разместить в памяти и инициализировать стек.
  DynStack(int size) {
    stck = new int[size];
    tos = -1;
  }
```

```
  // Поместить элемент в стек.
  public void push(int item) {
    // Если стек полон, тогда создать стек большего размера.
    if(tos==stck.length-1) {
      int[] temp = new int[stck.length * 2]; // удвоить размер
      for(int i=0; i<stck.length; i++) temp[i] = stck[i];
      stck = temp;
      stck[++tos] = item;
    }
    else
      stck[++tos] = item;
  }

  // Извлечь элемент из стека.
  public int pop() {
    if(tos < 0) {
      System.out.println("Стек опустошен.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class IFTest2 {
  public static void main(String[] args) {
    DynStack mystack1 = new DynStack(5);
    DynStack mystack2 = new DynStack(8);

    // Эти циклы заставляют увеличиваться каждый стек.
    for(int i=0; i<12; i++) mystack1.push(i);
    for(int i=0; i<20; i++) mystack2.push(i);

    System.out.println("Стек в mystack1:");
    for(int i=0; i<12; i++)
      System.out.println(mystack1.pop());

    System.out.println("Стек в mystack2:");
    for(int i=0; i<20; i++)
      System.out.println(mystack2.pop());
  }
}
```

В показанном далее классе используются обе реализации, `FixedStack` и `DynStack`, через ссылку на интерфейс. Другими словами, вызовы `push()` и `pop()` распознаются во время выполнения, а не на этапе компиляции.

```
/* Создать переменную ссылки на интерфейс и организовать
   через нее доступ к стекам.
*/
class IFTest3 {
  public static void main(String[] args) {
    IntStack mystack;    // создать переменную ссылки на интерфейс
    DynStack ds = new DynStack(5);
    FixedStack fs = new FixedStack(8);

    mystack = ds;        // загрузить в стек с динамическим размером
```

```
    // Поместить несколько чисел в стеки.
    for(int i=0; i<12; i++) mystack.push(i);

    mystack = fs;        // загрузить в стек с фиксированным размером
    for(int i=0; i<8; i++) mystack.push(i);

    mystack = ds;
    System.out.println("Значения в стеке с динамическим размером:");
    for(int i=0; i<12; i++)
      System.out.println(mystack.pop());

    mystack = fs;
    System.out.println("Значения в стеке с фиксированным размером:");
    for(int i=0; i<8; i++)
      System.out.println(mystack.pop());
  }
}
```

В этой программе `mystack` является ссылкой на интерфейс `IntStack`. Таким образом, когда она ссылается на `ds`, то используются версии `push()` и `pop()`, определенные реализацией `DynStack`, а когда на `fs`, то применяются версии `push()` и `pop()`, определенные в `FixedStack`. Как объяснялось ранее, распознавание производится во время выполнения. Доступ к множеству реализаций интерфейса через переменную ссылки на интерфейс — самый мощный способ обеспечения полиморфизма во время выполнения в Java.

Переменные в интерфейсах

Интерфейсы можно использовать для импортирования общих констант в несколько классов, просто объявив интерфейс, который содержит переменные, инициализированные нужными значениями. Когда такой интерфейс включается в класс (т.е. когда интерфейс "реализуется"), имена всех этих переменных будут находиться в области видимости как константы. Если интерфейс не содержит методов, то любой класс, включающий такой интерфейс, на самом деле ничего не реализует. Результат оказывается таким же, как если бы класс импортировал константные поля в пространство имен класса как переменные `final`. В следующем примере такой прием применяется для реализации автоматизированной "системы принятия решений":

```
import java.util.Random;

interface SharedConstants {
  int NO = 0;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}

class Question implements SharedConstants {
  Random rand = new Random();
  int ask() {
    int prob = (int) (100 * rand.nextDouble());
```

```
    if (prob < 30)
      return NO;              // 30%
    else if (prob < 60)
      return YES;             // 30%
    else if (prob < 75)
      return LATER;           // 15%
    else if (prob < 98)
      return SOON;            // 13%
    else
      return NEVER;           // 2%
  }
}
class AskMe implements SharedConstants {
  static void answer(int result) {
    switch(result) {
      case NO:
        System.out.println("Нет");
        break;
      case YES:
        System.out.println("Да");
        break;
      case MAYBE:
        System.out.println("Возможно");
        break;
      case LATER:
        System.out.println("Позже");
        break;
      case SOON:
        System.out.println("Скоро");
        break;
      case NEVER:
        System.out.println("Никогда");
        break;
    }
  }
  public static void main(String[] args) {
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
  }
}
```

Обратите внимание, что в программе используется один из стандартных классов Java — `Random`, который выдает псевдослучайные числа. Он содержит несколько методов, позволяющих получать случайные числа в том виде, который требуется вашей программе. В этом примере применяется метод `nextDouble()`, возвращающий случайные числа в диапазоне от `0.0` до `1.0`.

В приведенном примере программы два класса, `Question` и `AskMe`, реализуют интерфейс `SharedConstants`, в котором определены константные поля

NO, YES, MAYBE, SOON, LATER и NEVER. Внутри каждого класса код обращается к этим константам, как если бы каждый класс определял или наследовал их напрямую. Ниже показан вывод, полученный в результате запуска программы. Имейте в виду, что при каждом запуске результаты будут отличаться.

```
Позже
Скоро
Нет
Да
```

На заметку! Прием применения интерфейса для определения общих констант, как было описано выше, вызывает споры. Он рассмотрен здесь ради полноты.

Интерфейсы можно расширять

С помощью ключевого слова extends один интерфейс может быть унаследован от другого. Синтаксис используется такой же, как и при наследовании классов. Когда класс реализует интерфейс, унаследованный от другого интерфейса, он должен предоставить реализации для всех методов, требуемых интерфейсами в цепочке наследования. Ниже приведен пример:

```
// Один интерфейс может расширять другой.
interface A {
  void meth1();
  void meth2();
}

// Интерфейс B теперь включает meth1() и meth2() - он добавляет meth3().
interface B extends A {
  void meth3();
}

// Этот класс должен реализовать все методы интерфейсов A и B.
class MyClass implements B {
  public void meth1() {
    System.out.println("Реализация meth1().");
  }

  public void meth2() {
    System.out.println("Реализация meth2().");
  }

  public void meth3() {
    System.out.println("Реализация meth3().");
  }
}

class IFExtend {
  public static void main(String[] args) {
    MyClass ob = new MyClass();

    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```

В качестве эксперимента попробуйте удалить реализацию `meth1()` из `MyClass`. Результатом будет ошибка на этапе компиляции. Как уже упоминалось, любой класс, реализующий интерфейс, должен реализовывать все методы, требуемые этим интерфейсом, в том числе все методы, унаследованные от других интерфейсов.

Стандартные методы интерфейса

Как объяснялось ранее, до выхода JDK 8 интерфейс не мог определять какую-либо реализацию. Таким образом, во всех предшествующих версиях Java методы, определяемые интерфейсом, были абстрактными и не содержали тела. Это традиционная форма интерфейса, и именно такой вид интерфейса применялся в предыдущих обсуждениях. В выпуске JDK 8 ситуация изменилась за счет добавления к интерфейсам новой возможности, которая называется *стандартным методом*. Стандартный метод позволяет определить реализацию по умолчанию для метода интерфейса. Другими словами, используя стандартный метод, метод интерфейса может предоставлять тело, а не быть абстрактным. Во время разработки стандартный метод также упоминался как *расширяющий метод* и вполне вероятно, что вы столкнетесь с обоими терминами.

Основным мотивом ввода стандартного метода было предоставление средств, с помощью которых удалось бы расширять интерфейсы без нарушения работы существующего кода. Вспомните, что должны быть предусмотрены реализации для всех методов, определенных интерфейсом. В прошлом добавление нового метода к популярному, широко применяемому интерфейсу нарушало работу существующего кода, поскольку для нового метода отсутствовала реализация. Проблему решает стандартный метод, предоставляя реализацию, которая будет использоваться, если явно не указана другая реализация. Таким образом, добавление стандартного метода не приводит к нарушению работы существующего кода.

Еще одним мотивом ввода стандартного метода было желание указывать в интерфейсе методы, которые по существу являются необязательными в зависимости от того, как применяется интерфейс. Скажем, интерфейс может определять группу методов, которые воздействуют на последовательность элементов. Один из этих методов может называться `remove()`, и его цель — удаление элемента из последовательности. Однако если интерфейс предназначен для поддержки как изменяемых, так и неизменяемых последовательностей, то метод `remove()` не считается обязательным, т.к. он не будет использоваться неизменяемыми последовательностями. В прошлом класс, который реализовывал неизменяемую последовательность, должен был определять пустую реализацию метода `remove()`, даже если в нем не было необходимости. Теперь в интерфейсе может быть указана стандартная реализация для `remove()`, которая ничего не делает (или генерирует исключение). Предоставление стандартной реализации делает необязательным определение в классе, предназначенном для неизменяемых последовательностей, собственной версии-заглушки метода

remove(). Таким образом, за счет обеспечения стандартного метода реализация remove() в классе становится необязательной.

Важно отметить, что добавление стандартных методов не меняет ключевой аспект интерфейса: его неспособность поддерживать информацию о состоянии. Например, интерфейс по-прежнему не может иметь переменные экземпляра. Следовательно, определяющее различие между интерфейсом и классом заключается в том, что класс может поддерживать информацию о состоянии, а интерфейс — нет. Кроме того, по-прежнему нельзя создавать экземпляр интерфейса самого по себе. Интерфейс должен быть реализован классом. По этой причине, несмотря на появившуюся в версии JDK 8 возможность определения стандартных методов, интерфейс все равно должен быть реализован классом, если необходимо создавать экземпляр.

И последнее замечание: как правило, стандартные методы представляют собой функциональное средство специального назначения. Интерфейсы, которые вы создаете, по-прежнему будут применяться в основном для указания того, *что* делать, а не *как*. Тем не менее, появление стандартных методов обеспечивает дополнительную гибкость.

Основы стандартных методов

Стандартный метод интерфейса определяется аналогично определению метода в классе. Основное отличие связано с тем, что объявление предваряется ключевым словом default. Например, возьмем следующий простой интерфейс:

```
public interface MyIF {
  // Это объявление "нормального" метода интерфейса.
  // В нем НЕ определяется стандартная реализация.
  int getNumber();

  // Это стандартный метод. Обратите внимание,
  // что он предоставляет реализацию по умолчанию.
  default String getString() {
    return "Стандартная строка";
  }
}
```

В интерфейсе MyIF объявлены два метода. Объявление первого, getNumber(), является объявлением обычного метода интерфейса. Какая-либо реализация в нем отсутствует. Второй метод, getString(), включает стандартную реализацию. В данном случае он просто возвращает строку "Стандартная строка". Обратите особое внимание на способ объявления getString(). Его объявлению предшествует модификатор default. Такой синтаксис можно обобщить. Чтобы определить стандартный метод, перед его объявлением необходимо указать default.

Поскольку getString() содержит стандартную реализацию, реализующему классу переопределять ее необязательно. Другими словами, если реализующий класс не предоставляет собственную реализацию, то используется стандартная. Скажем, показанный ниже класс MyIFImp совершенно допустим:

```
// Реализовать MyIF.
class MyIFImp implements MyIF {
  // Необходимо реализовать только метод getNumber(), определенный в MyIF.
  // Для метода getString() разрешено применять стандартную реализацию.
  public int getNumber() {
    return 100;
  }
}
```

В следующем коде создается экземпляр класса `MyIFImp`, который применяется для вызова методов `getNumber()` и `getString()`:

```
// Использовать стандартный метод.
class DefaultMethodDemo {
  public static void main(String[] args) {

    MyIFImp obj = new MyIFImp();

    // Метод getNumber() можно вызывать, потому что
    // он явно реализован в MyIFImp:
    System.out.println(obj.getNumber());

    // Метод getString() тоже можно вызывать
    // из-за наличия стандартной реализации:
    System.out.println(obj.getString());
  }
}
```

Вот вывод, генерируемый программой:

```
100
Стандартная строка
```

Как видите, автоматически была использована стандартная реализация метода `getString()`. Определять ее в классе `MyIFImp` не понадобилось. Таким образом, для метода `getString()` реализация классом необязательна. (Конечно, реализовать его в классе *потребуется*, если класс применяет `getString()` для каких-либо целей помимо тех, что поддерживаются стандартной реализацией.)

Для реализующего класса возможно и распространено определение собственной реализации стандартного метода. Например, в класс `MyIFImp2` метод `getString()` переопределяется:

```
class MyIFImp2 implements MyIF {
  // Здесь предоставляются реализации для обоих методов, getNumber()
  // и getString().
  public int getNumber() {
    return 100;
  }

  public String getString() {
    return "Другая строка.";
  }
}
```

Теперь при вызове `getString()` возвращается другая строка.

Более реалистичный пример

Хотя в рассмотренных выше примерах демонстрировалась механика использования стандартных методов, не была проиллюстрирована их полезность в более реалистичных условиях. Давайте еще раз вернемся к интерфейсу `IntStack`, показанному ранее в главе. В целях обсуждения предположим, что `IntStack` широко применяется, и на него полагаются многие программы. Кроме того, пусть теперь мы хотим добавить в `IntStack` метод, который очищает стек, позволяя использовать его многократно. Таким образом, мы желаем развивать интерфейс `IntStack`, чтобы он определял новые функции, но не хотим нарушать работу уже существующего кода. В прошлом решить задачу было невозможно, но с включением стандартных методов теперь это сделать легко. Например, интерфейс `IntStack` можно расширить следующим образом:

```
interface IntStack {
  void push(int item);   // сохранить элемент
  int pop();             // извлечь элемент

  // Поскольку метод clear() имеет стандартную реализацию, его не нужно
  // реализовывать в существующем классе, использующем IntStack.
  default void clear() {
    System.out.println("Метод clear() не реализован.");
  }
}
```

Стандартное поведение `clear()` предусматривает просто отображение сообщения о том, что метод не реализован. Поступать так допустимо, поскольку ни один существующий класс, который реализует `IntStack`, не мог вызывать метод `clear()`, потому что он не был определен в более ранней версии `IntStack`. Однако метод `clear()` можно реализовать в новом классе, реализующем `IntStack`. Кроме того, метод `clear()` необходимо определять с новой реализацией только в том случае, если он задействован. В итоге стандартный метод предоставляет:

- способ элегантного развития интерфейсов с течением времени;
- способ обеспечения дополнительной функциональности без требования, чтобы класс предоставлял реализацию заглушки, когда эта функциональность не нужна.

И еще один момент: в реальном коде метод `clear()` инициировал бы исключение, а не отображал сообщение об ошибке. Исключения описаны в следующей главе. После изучения этого материала вы можете попробовать изменить метод `clear()`, чтобы его стандартная реализация выдавала исключение `UnsupportedOperationException`.

Проблемы множественного наследования

Как объяснялось ранее в книге, язык Java не поддерживает множественное наследование классов. Теперь, когда интерфейс способен содержать

стандартные методы, вам может быть интересно, удастся ли с помощью интерфейсов обойти это ограничение. Ответ: по существу нет. Вспомните, что между классом и интерфейсом все же есть ключевое различие: класс может поддерживать информацию о состоянии (особенно за счет использования переменных экземпляра), а интерфейс — нет.

Тем не менее, стандартные методы предлагают то, что обычно ассоциируется с концепцией множественного наследования. Скажем, у вас может быть класс, реализующий два интерфейса. Если каждый из таких интерфейсов предоставляет стандартные методы, то некоторое поведение наследуется от обоих. Таким образом, стандартные методы в ограниченной степени поддерживают множественное наследование поведения. Как несложно догадаться, в подобной ситуации возможен конфликт имен.

Например, пусть класс `MyClass` реализует два интерфейса с именами `Alpha` и `Beta`. Что произойдет, если и `Alpha`, и `Beta` предоставят метод `reset()`, для которого оба интерфейса объявят стандартную реализацию? Какая версия `reset()` будет задействована в классе `MyClass` — из интерфейса `Alpha` либо из интерфейса `Beta`? Или рассмотрим ситуацию, в которой интерфейс `Beta` расширяет `Alpha`. Какая версия стандартного метода используется? А что, если `MyClass` предоставляет собственную реализацию метода? Для обработки этих и других похожих ситуаций в Java определен набор правил, разрешающих такие конфликты.

Во-первых, во всех случаях реализация класса имеет приоритет над стандартной реализацией интерфейса. Таким образом, если в `MyClass` переопределяется стандартный метод `reset()`, то применяется версия из `MyClass`. Это так, даже если `MyClass` реализует и `Alpha`, и `Beta`. Тогда оба стандартных метода переопределяются реализацией в `MyClass`.

Во-вторых, в случаях, когда класс реализует два интерфейса с одним и тем же стандартным методом, но класс не переопределяет этот метод, возникает ошибка. Продолжая пример, если `MyClass` реализует и `Alpha`, и `Beta`, но не переопределяет `reset()`, тогда произойдет ошибка.

В случаях, когда один интерфейс унаследован от другого, и оба определяют общий стандартный метод, приоритет имеет версия метода из наследующего интерфейса. Поэтому, продолжая пример, если `Beta` расширяет `Alpha`, то будет использоваться версия `reset()` из `Beta`.

С применением формы `super` следующего вида в унаследованном интерфейсе можно явно ссылаться на стандартную реализацию:

ИмяИнтерфейса`.super.`*имяМетода*`()`

Например, если в `Beta` нужно сослаться на стандартный метод `reset()` из `Alpha`, то вот какой оператор можно применить:

```
Alpha.super.reset();
```

Использование статических методов в интерфейсе

В версии JDK 8 в интерфейсах появилась возможность определения одного или нескольких статических методов. Подобно статическим методам в классе статический метод, определенный в интерфейсе, может вызываться независимо от любого объекта. Таким образом, для вызова статического метода не требуется реализация интерфейса и экземпляр реализации интерфейса. Взамен статический метод вызывается путем указания имени интерфейса, за которым следует точка и имя метода. Вот общая форма:

ИмяИнтерфейса.имяСтатическогоМетода()

Обратите внимание сходство со способом вызова статического метода в классе.

Ниже показан пример статического метода, добавленного в интерфейс `MyIF` из предыдущего раздела. Статический метод `getDefaultNumber()` возвращает значение 0.

```
public interface MyIF {
  // Это объявление "нормального" метода интерфейса.
  // В нем НЕ определяется стандартная реализация.
  int getNumber();

  // Это стандартный метод. Обратите внимание,
  // что он предоставляет стандартную реализацию.
  default String getString() {
    return "Стандартная строка";
  }

  // Это статический метод интерфейса.
  static int getDefaultNumber() {
    return 0;
  }
}
```

Метод `getDefaultNumber()` может быть вызван следующим образом:

```
int defNum = MyIF.getDefaultNumber();
```

Как уже упоминалось, для вызова метода `getDefaultNumber()` не требуется никакой реализации или экземпляра реализации `MyIF`, потому что он является статическим.

И последнее замечание: статические методы интерфейса не наследуются ни реализующим классом, ни производными интерфейсами.

Закрытые методы интерфейса

Начиная с версии JDK 9, интерфейс способен содержать закрытый метод, который может вызываться только стандартным методом или другим закрытым методом, определенным в том же интерфейсе. Поскольку закрытый метод интерфейса указан как `private`, его нельзя использовать в коде вне

интерфейса, в котором он определен. Такое ограничение распространяется и на производные интерфейсы, потому что закрытый метод интерфейса ими не наследуется.

Основное преимущество закрытого метода интерфейса заключается в том, что он позволяет двум или более стандартным методам использовать общий фрагмент кода, позволяя избежать дублирования кода. Например, ниже показана еще одна версия интерфейса `IntStack`, которая имеет два стандартных метода `popNElements()` и `skipAndPopNElements()`. Первый метод возвращает массив, содержащий *N* верхних элементов стека. Второй метод пропускает указанное количество элементов, а затем возвращает массив, содержащий следующие *N* элементов. Оба метода применяют закрытый метод `getElements()` для получения массива с указанным количеством элементов из стека.

```
// Еще одна версия IntStack, имеющая закрытый метод интерфейса,
// который используется двумя стандартными методами.

interface IntStack {
  void push(int item);       // сохранить элемент
  int pop();                 // извлечь элемент

  // Стандартный метод, возвращающий массив, который
  // содержит верхние n элементов в стеке.
  default int[] popNElements(int n) {
    // Возвратить запрошенные элементы.
    return getElements(n);
  }

  // Стандартный метод, возвращающий массив, который содержит
  // следующие n элементов в стеке после пропуска skip элементов.
  default int[] skipAndPopNElements(int skip, int n) {

    // Пропустить указанное количество элементов.
    getElements(skip);

    // Возвратить запрошенные элементы.
    return getElements(n);
  }

  // Закрытый метод, который возвращает массив, содержащий
  // верхние n элементов в стеке.
  private int[] getElements(int n) {
    int[] elements = new int[n];

    for(int i=0; i < n; i++) elements[i] = pop();
    return elements;
  }
}
```

Обратите внимание, что для получения возвращаемого массива в `popNElements()` и `skipAndPopNElements()` применяется закрытый метод `getElements()`. Это предотвращает дублирование одной и той же кодовой последовательности в обоих методах. Имейте в виду, что поскольку метод `getElements()` является закрытым, его нельзя вызывать из кода за пределами `IntStack`. Таким образом, его использование ограничено стандартными

методами внутри IntStack. Кроме того, из-за того, что для получения элементов стека в getElements() применяется метод pop(), он автоматически вызывает реализацию pop(), предоставляемую реализацией IntStack. Следовательно, getElements() будет работать для любого класса стека, реализующего IntStack.

Хотя закрытый метод интерфейса представляет собой функциональное средство, которое будет востребовано редко, в тех случаях, когда оно вам нужно, вы найдете его весьма полезным.

Заключительные соображения по поводу пакетов и интерфейсов

Хотя в примерах, включенных в книгу, пакеты или интерфейсы используются не особенно часто, оба эти инструмента являются важной частью программной среды Java. Практически все реальные программы, которые придется писать на Java, будут содержаться в пакетах. Некоторые из них, вероятно, также будут реализовывать интерфейсы. Поэтому важно, чтобы вы научились их применять надлежащим образом.

ГЛАВА

10 Обработка исключений

В настоящей главе рассматривается механизм обработки исключений Java. *Исключение* — это ненормальное состояние, которое возникает в кодовой последовательности во время выполнения. Другими словами, исключение является ошибкой времени выполнения. В языках программирования, не поддерживающих обработку исключений, ошибки необходимо проверять и обрабатывать вручную — обычно с помощью кодов ошибок и т.д. Такой подход столь же громоздкий, сколь и хлопотный. Обработка исключений в Java позволяет избежать проблем подобного рода и попутно переносит управление ошибками во время выполнения в объектно-ориентированный мир.

Основы обработки исключений

Исключение Java представляет собой объект, описывающий исключительное (т.е. ошибочное) состояние, которое произошло внутри фрагмента кода. При возникновении исключительного состояния в методе, вызвавшем ошибку, *генерируется* объект, представляющий это исключение. Метод может обработать исключение самостоятельно или передать его дальше. Так или иначе, в какой-то момент исключение *перехватывается* и *обрабатывается*. Исключения могут быть сгенерированы исполняющей средой Java или вручную в вашем коде. Исключения, генерируемые Java, относятся к фундаментальным ошибкам, которые нарушают правила языка Java или ограничения исполняющей среды Java. Исключения, сгенерированные вручную, обычно используются для сообщения об ошибке вызывающей стороне метода.

Обработка исключений в Java управляется пятью ключевыми словами: `try`, `catch`, `throw`, `throws` и `finally`. Давайте кратко рассмотрим, как они работают. Операторы программы, которые вы хотите отслеживать на наличие исключений, содержатся в блоке `try`. Если внутри блока `try` возникает исключение, тогда оно генерируется. Ваш код может перехватить это исключение (с помощью `catch`) и обработать его рациональным образом. Системные исключения автоматически генерируются исполняющей средой Java. Для ручной генерации исключения используйте ключевое слово `throw`. Любое исключение, генерируемое в методе, должно быть указано как таковое с помощью конструкции `throws`. Любой код, который обязательно должен быть выполнен после завершения блока `try`, помещается в блок `finally`.

Ниже показана общая форма блока обработки исключений:

```
try {
  // блок кода, где отслеживаются ошибки
}
catch (ТипИсключения1 объектИсключения) {
  // обработчик исключений для ТипИсключения1
}
catch (ТипИсключения2 объектИсключения) {
  // обработчик исключений для ТипИсключения2
}
// ...
finally {
  // блок кода, подлежащий выполнению после окончания блока try
}
```

Здесь *ТипИсключения* — это тип возникшей исключительной ситуации. В оставшихся материалах главы демонстрируется применение приведенной выше структуры.

На заметку! Существует еще одна форма оператора `try`, которая поддерживает *автоматическое управление ресурсами*. Она называется *`try` с ресурсами* и описана в главе 13 в контексте управления файлами, поскольку файлы являются одним из наиболее часто используемых ресурсов.

Типы исключений

Все типы исключений являются подклассами встроенного класса `Throwable`. Таким образом, класс `Throwable` расположен на вершине иерархии классов исключений. Непосредственно под `Throwable` находятся два подкласса, которые разделяют исключения на две отдельные ветви. Одну ветвь возглавляет класс `Exception`, используемый для представления исключительных условий, которые должны перехватываться пользовательскими программами. Он также будет служить подклассом для создания собственных специальных типов исключений. Кроме того, у класса `Exception` имеется важный подкласс, который называется `RuntimeException`. Исключения такого типа автоматически определяются для разрабатываемых программ и охватывают такие ситуации, как деление на ноль и недопустимое индексирование массивов.

Другую ветвь возглавляет класс `Error`, определяющий исключения, которые не должны перехватываться программой в обычных условиях. Исключения типа `Error` применяется исполняющей средой Java для указания ошибок, связанных с самой средой. Примером такой ошибки является переполнение стека. Исключения типа `Error` здесь не рассматриваются, т.к. они обычно создаются в ответ на катастрофические отказы, которые обычно не могут быть обработаны создаваемой программой.

Иерархия исключений верхнего уровня показана на рис. 10.1.

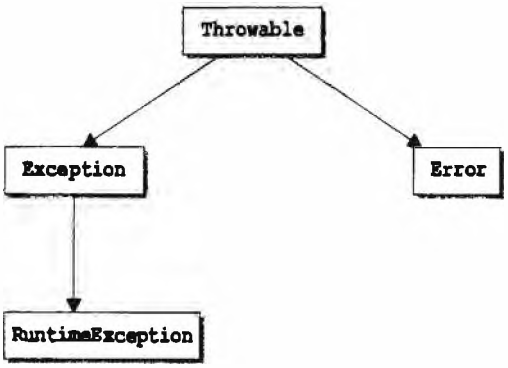


Рис. 10.1. Иерархия исключений верхнего уровня

Неперехваченные исключения

Прежде чем вы научитесь обрабатывать исключения в своей программе, полезно посмотреть, что происходит, когда их не обрабатывать. Следующая небольшая программа содержит выражение, которое намеренно вызывает ошибку деления на ноль:

```
class Exc0 {
  public static void main(String[] args) {
    int d = 0;
    int a = 42 / d;
  }
}
```

Когда исполняющая среда Java обнаруживает попытку деления на ноль, она создает новый объект исключения и затем *генерирует* это исключение. В результате выполнение класса `Exc0` останавливается, поскольку после генерации исключение должно быть *перехвачено* обработчиком исключений и немедленно обработано. В приведенном примере не было предусмотрено никаких собственных обработчиков исключений, поэтому исключение перехватывается стандартным обработчиком, предоставляемым исполняющей средой Java. Любое исключение, которое не перехвачено вашей программой, в конечном итоге будет обработано стандартным обработчиком. Стандартный обработчик отображает строку с описанием исключения, выводит трассировку стека от точки, где произошло исключение, и прекращает работу программы.

Вот какое исключение генерируется при выполнении примера:

```
java.lang.ArithmeticException: / by zero
    at Exc0.main(Exc0.java:4)
java.lang.ArithmeticException: деление на ноль
    в Exc0.main(Exc0.java:4)
```

Обратите внимание, что в простой трассировки стека присутствует имя класса — `Exc0`, имя метода — `main()`, имя файла — `Exc0.java` и номер строки — 4. Кроме того, как видите, типом сгенерированного исключения является подкласс `Exception` по имени `ArithmeticException`, который более

конкретно описывает тип возникшей ошибки. Далее в главе будет показано, что Java предлагает несколько встроенных типов исключений, соответствующих различным типам ошибок времени выполнения, которые могут быть сгенерированы. Еще одно замечание: точный вывод, который вы видите при запуске этого и других примеров программ в главе, использующих встроенные исключения Java, может немного отличаться от показанного здесь из-за различий между версиями JDK.

В трассировке стека всегда показана последовательность вызовов методов, которые привели к ошибке. Например, вот еще одна версия предыдущей программы, которая вызывает ту же ошибку, но в методе, отдельном от `main()`:

```
class Excl {
  static void subroutine() {
    int d = 0;
    int a = 10 / d;
  }
  public static void main(String[] args) {
    Excl.subroutine();
  }
}
```

В полученной трассировке стека из стандартного обработчика исключений видно, что отображается весь стек вызовов:

```
java.lang.ArithmeticException: / by zero
    at Excl.subroutine(Excl.java:4)
    at Excl.main(Excl.java:7)
```

Как видите, в нижней части стека указана строка 7 метода `main()`, в которой вызывается метод `subroutine()`, ставший причиной исключения в строке 4. Стек вызовов весьма полезен для отладки, поскольку он указывает точную последовательность шагов, которые привели к ошибке.

Использование `try` и `catch`

Хотя стандартный обработчик исключений, предоставляемый исполняющей средой Java, удобен при отладке, обычно вы пожелаете обрабатывать исключение самостоятельно, что дает два преимущества. Самостоятельная обработка, во-первых, позволяет исправить ошибку и, во-вторых, предотвращает автоматическое прекращение работы программы. Большинство пользователей будут (по меньшей мере) сбиты с толку, если ваша программа перестанет работать, начав выводить трассировку стека всякий раз, когда возникает ошибка! К счастью, предотвратить это довольно легко.

Чтобы защититься от ошибки времени выполнения и обработать ее, просто поместите код, который хотите отслеживать, в блок `try`. Сразу после блока `try` добавьте конструкцию `catch` с указанием типа исключения, которое желательно перехватить. В целях иллюстрации того, насколько легко это делать, в следующей программе определен блок `try` и конструкция `catch`, обрабатывающая исключение `ArithmeticException`, которое генерируется ошибкой деления на ноль:

```
class Exc2 {
public static void main(String[] args) {
  int d, a;

  try {  // отслеживать блок кода
    d = 0;
    a = 42 / d;
    System.out.println("Это выводиться не будет.");
    } catch (ArithmeticException e) {  // перехватить ошибку деления на ноль
      System.out.println("Деление на ноль.");
    }
    System.out.println("После оператора catch.");
  }
}
```

Программа выдает такой вывод:

```
Деление на ноль.
После оператора catch.
```

Обратите внимание, что вызов println() внутри блока try никогда не выполняется. После генерации исключения управление передается из блока try в блок catch. Другими словами, блок catch не "вызывается" и потому управление никогда не "возвращается" в блок try из catch. Таким образом, строка "Это выводиться не будет." не отображается. После блока catch выполнение продолжается со строки программы, следующей за всем механизмом try/catch.

Оператор try и его конструкция catch образуют единицу. Область действия catch ограничена операторами, которые относятся к непосредственно предшествующему оператору try. Конструкция catch не может перехватывать исключение, сгенерированное другим оператором try (за исключением описанного ниже случая вложенных операторов try). Операторы, защищенные с помощью try, должны быть заключены в фигурные скобки (т.е. находиться внутри блока). Применять try для одиночного оператора нельзя.

Целью большинства хорошо построенных конструкций catch должно быть разрешение исключительной ситуации и продолжение работы, как если бы ошибка вообще не возникала. Например, в приведенной далее программе на каждой итерации цикла for получаются два случайных целых числа, одно из которых делится на другое, а результат используется для деления значения 12345. Окончательный результат помещается в переменную a. Если какая-либо операция вызывает ошибку деления на ноль, то она перехватывается, значение a устанавливается равным нулю и выполнение программы продолжается.

```
// Обработать исключение и продолжить работу.
import java.util.Random;

class HandleError {
  public static void main(String[] args) {
    int a=0, b=0, c=0;
    Random r = new Random();
```

```
    for(int i=0; i<32000; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Деление на ноль.");
        a = 0;  // установить a в ноль и продолжить
      }
      System.out.println("a: " + a);
    }
  }
}
```

Отображение описания исключения

В классе `Throwable` переопределен метод `toString()` (определенный в `Object`), так что он возвращает строку, содержащую описание исключения. Для отображения этого описания в операторе `println()` нужно просто передать исключение в качестве аргумента. Например, блок `catch` из предыдущей программы можно переписать так:

```
catch (ArithmeticException e) {
  System.out.println("Исключение: " + e);
  a = 0;  // установить a в ноль и продолжить
}
```

Тогда в случае ошибки деления на ноль будет отображаться следующее сообщение:

```
Исключение: java.lang.ArithmeticException: / by zero
```

Хотя в таком контексте это не имеет особой ценности, возможность отображать описание исключения полезна в других обстоятельствах, особенно когда вы экспериментируете с исключениями или занимаетесь отладкой.

Использование нескольких конструкций `catch`

В некоторых случаях один фрагмент кода может генерировать более одного исключения. Чтобы справиться с ситуацией такого рода, можно указать две или более конструкции `catch`, каждая из которых будет перехватывать разные типы исключений. При возникновении исключения все конструкции `catch` проверяются по порядку, и выполняется первая из них, в которой указанный тип совпадает с типом сгенерированного исключения. После выполнения одной конструкции `catch` остальные игнорируются, и выполнение продолжается после блока `try/catch`. В следующем примере перехватываются два разных типа исключений:

```
// Демонстрация применения нескольких конструкций catch.
class MultipleCatches {
  public static void main(String[] args) {
    try {
```

```
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int[] c = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Деление на ноль: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
     System.out.println("Выход за допустимые пределы индекса в массиве: " + e);
    }
    System.out.println("После блоков try/catch.");
  }
}
```

Программа вызовет исключение деления на ноль, если будет запущена без аргументов командной строки, т.к. значение a будет равно нулю. Деление пройдет успешно в случае предоставления аргумента командной строки, который приведет к установке a во что-то большее, чем ноль. Но это станет причиной генерации исключения `ArrayIndexOutOfBoundsException`, поскольку целочисленный массив c имеет длину 1, а программа пытается присвоить значение несуществующему элементу `c[42]`.

Ниже показан вывод программы, выдаваемый в обеих ситуациях:

```
C:\>java MultipleCatches a = 0
Деление на ноль: java.lang.ArithmeticException: / by zero
После блоков try/catch.

C:\>java MultipleCatches TestArg a = 1
Выход за допустимые пределы индекса в массиве:
java.lang.ArrayIndexOutOfBoundsException:
Index 42 out of bounds for length 1
После блоков try/catch.
```

При использовании нескольких конструкций `catch` важно помнить о том, что подклассы исключений должны предшествовать любым из своих суперклассов. Дело в том, что конструкция `catch`, в которой применяется суперкласс, будет перехватывать исключения указанного типа плюс любых его подклассов. В итоге конструкция `catch` с подклассом никогда не будет достигнута, если она находится после конструкции `catch` с суперклассом. Кроме того, недостижимый код в Java является ошибкой. Например, рассмотрим следующую программу:

```
/* Эта программа содержит ошибку.

   В последовательности конструкций catch подкласс должен
   предшествовать своему суперклассу. В противном случае будет создан
   недостижимый код, что приведет к ошибке на этапе компиляции.
*/
class SuperSubCatch {
  public static void main(String[] args) {
    try {
      int a = 0;
      int b = 42 / a;
```

```
    } catch(Exception e) {
      System.out.println("Перехват обобщенного исключения Exception.");
    }
    /* Эта конструкция catch недостижима, потому что
       ArithmeticException является подклассом Exception. */
    catch(ArithmeticException e) {  // ОШИБКА - недостижимый код
      System.out.println("Это никогда не будет достигнуто.");
    }
  }
}
```

Попытка компиляции этой программы приводит к получению сообщения об ошибке, указывающего на то, что вторая конструкция `catch` недостижима, т.к. исключение уже было перехвачено. Поскольку `ArithmeticException` является подклассом `Exception`, первая конструкция `catch` будет обрабатывать все ошибки, связанные с `Exception`, в том числе `ArithmeticException`. Таким образом, вторая конструкция `catch` никогда не выполнится. Чтобы решить проблему, понадобится изменить порядок следования конструкций `catch`.

Вложенные операторы `try`

Оператор `try` может быть вложенным, т.е. находиться внутри блока другого оператора `try`. Каждый раз, когда происходит вход в `try`, контекст этого исключения помещается в стек. Если внутренний оператор `try` не имеет обработчика `catch` для определенного исключения, тогда стек раскручивается, и на предмет совпадения проверяются обработчики `catch` следующего оператора `try`. Процесс продолжается до тех пор, пока не будет найдена подходящая конструкция `catch` либо исчерпаны все вложенные операторы `try`. Если ни одна из конструкций `catch` не дает совпадения, то исключение будет обработано исполняющей средой Java. Ниже приведен пример использования вложенных операторов `try`:

```
// Пример применения вложенных операторов try.
class NestTry {
  public static void main(String[] args) {
    try {
      int a = args.length;

      /* Если аргументы командной строки отсутствуют, то следующий
         оператор сгенерирует исключение деления на ноль. */
      int b = 42 / a;

      System.out.println("a = " + a);

      try {  // вложенный блок try
        /* Если используется один аргумент командной строки, тогда
           исключение деления на ноль сгенерирует следующий код. */
        if(a==1) a = a/(a-a);  // деление на ноль

        /* Если используется один аргумент командной строки,
           тогда генерируется исключение выхода за допустимые
           пределы индекса в массиве. */
```

```
        if(a==2) {
          int[] c = { 1 };
          c[42] = 99;  // генерирует исключение ArrayIndexOutOfBoundsException
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Выход за допустимые пределы индекса в массиве: "+ e);
      }
    } catch(ArithmeticException e) {
      System.out.println("Деление на ноль: " + e);
    }
  }
}
```

Как видите, в программе один блок try вложен в другой. Программа работает следующим образом. При запуске программы без аргументов командной строки внешний блок try генерирует исключение деления на ноль. Запуск программы с одним аргументом командной строки приводит к генерации исключения деления на ноль внутри вложенного блока try. Поскольку внутренний блок try не перехватывает это исключение, оно передается внешнему блоку try, где и обрабатывается. При запуске программы с двумя аргументами командной строки во внутреннем блоке try генерируется исключение выхода за допустимые пределы индекса в массиве. Вот примеры запуска, иллюстрирующие каждый случай:

```
C:\>java NestTry
Деление на ноль: java.lang.ArithmeticException: / by zero

C:\>java NestTry One a = 1
Деление на ноль: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two a = 2
Выход за допустимые пределы индекса в массиве:
java.lang.ArrayIndexOutOfBoundsException:
Index 42 out of bounds for length 1
```

Когда задействованы вызовы методов, вложение операторов try может происходить менее очевидным образом. Например, если вызов метода заключен в блок try и внутри этого метода находится еще один оператор try, то оператор try внутри метода будет вложен во внешний блок try, где метод вызывается. Ниже представлена предыдущая программа, в которой вложенный блок try перемещен внутрь метода nesttry():

```
/* Операторы try могут быть неявно вложенными через вызовы методов. */
class MethNestTry {
  static void nesttry(int a) {
    try {  // вложенный блок try
      /* Если используется один аргумент командной строки, тогда
         исключение деления на ноль сгенерирует следующий код. */
      if(a==1) a = a/(a-a);  // деление на ноль

      /* Если используются два аргумента командной строки, тогда генерируется
         исключение выхода за допустимые пределы индекса в массиве. */
      if(a==2) {
        int[] c = { 1 };
```

```
      c[42] = 99;  // генерирует исключение ArrayIndexOutOfBoundsException
    }
  } catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Выход за допустимые пределы индекса в массиве: " + e);
  }
 }

 public static void main(String[] args) {
  try {
    int a = args.length;

    /* Если аргументы командной строки отсутствуют, то следующий
       оператор сгенерирует исключение деления на ноль. */
    int b = 42 / a;
    System.out.println("a = " + a);
    nesttry(a);
  } catch(ArithmeticException e) {
     System.out.println("Деление на ноль: " + e);
  }
 }
}
```

Вывод программы идентичен выводу в предыдущем примере.

Оператор throw

До сих пор перехватывались только те исключения, которые генерируются исполняющей средой Java. Однако программа может генерировать исключение явно с применением оператора `throw` со следующей общей формой:

```
throw ThrowableInstance;
```

Здесь `ThrowableInstance` должен быть объектом типа `Throwable` или подклассом `Throwable`. Примитивные типы вроде `int` или `char`, а также классы, отличающиеся от `Throwable`, такие как `String` и `Object`, не могут использоваться в качестве исключений. Есть два способа получить объект `Throwable`: указывая параметр в конструкции `catch` или создавая его с помощью операции `new`.

Поток выполнения останавливается сразу после оператора `throw`; любые последующие операторы не выполняются. Ближайший охватывающий блок `try` проверяется на предмет наличия в нем конструкции `catch`, соответствующей типу исключения. Если совпадение найдено, то управление передается этому оператору, а если нет, тогда проверяется следующий охватывающий оператор `try` и т.д. Если соответствующая конструкция `catch` не найдена, то стандартный обработчик исключений останавливает работу программы и выводит трассировку стека.

Ниже приведен пример программы, в которой создается и генерируется исключение. Обработчик, перехватывающий исключение, повторно генерирует его для внешнего обработчика.

```
// Демонстрация применения throw.
class ThrowDemo {
```

```
  static void demoproc() {
    try {
      throw new NullPointerException("демонстрация");
    } catch(NullPointerException e) {
      System.out.println("Перехвачено внутри demoproc().");
      throw e;          // повторно сгенерировать исключение
    }
  }
  public static void main(String[] args) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Повторно перехвачено: " + e);
    }
  }
}
```

В программе есть два шанса справиться с одной и той же ошибкой. Сначала в `main()` устанавливается контекст исключения и вызывается `demoproc()`. Затем метод `demoproc()` устанавливает другой контекст обработки исключений и немедленно создает новый экземпляр `NullPointerException`, который перехватывается в следующей строке. Далее исключение генерируется повторно. Вот результат:

```
Перехвачено внутри demoproc().
Повторно перехвачено: java.lang.NullPointerException: демонстрация
```

В программе также демонстрируется создание одного из стандартных объектов исключений Java. Обратите особое внимание на следующую строку:

```
throw new NullPointerException("демонстрация");
```

Здесь с применением операции `new` создается экземпляр `NullPointerException`. Многие встроенные исключения времени выполнения Java имеют как минимум два конструктора: один без параметров и один принимающий строковый параметр. Когда используется вторая форма, аргумент указывает строку, описывающую исключение. Эта строка отображается, когда объект передается как аргумент в `print()` или `println()`. Его также можно получить, вызвав метод `getMessage()`, который определен в `Throwable`.

Конструкция throws

Если метод способен приводить к исключению, которое он не обрабатывает, то метод должен сообщить о таком поведении, чтобы вызывающий его код мог защитить себя от этого исключения. Задача решается добавлением к объявлению метода конструкции `throws`, где перечисляются типы исключений, которые может генерировать метод. Поступать так необходимо для всех исключений, кроме исключений типа `Error`, `RuntimeException` или любых их подклассов. Все остальные исключения, которые может генерировать метод, должны быть объявлены в конструкции `throws`. В противном случае возникнет ошибка на этапе компиляции.

Вот общая форма объявления метода, которая содержит конструкцию throws:

```
тип имя-метода(список-параметров) throws список-исключений
{
  // тело метода
}
```

Здесь *список-исключений* представляет собой список разделяемых запятыми исключений, которые метод может сгенерировать.

Ниже приведен пример некорректной программы, пытающейся сгенерировать исключение, которое она не перехватывает. Из-за того, что в программе не указана конструкция throws для объявления данного факта, программа не скомпилируется.

```
// Эта программа содержит ошибку и компилироваться не будет.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Внутри throwOne().");
    throw new IllegalAccessException("демонстрация");
  }

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

Чтобы пример скомпилировался, в него понадобится внести два изменения. Во-первых, вам нужно объявить, что метод throwOne() генерирует исключение IllegalAccessException. Во-вторых, в методе main() должен быть определен оператор try/catch, который перехватывает это исключение.

Далее показан исправленный пример:

```
// Теперь программа компилируется.
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Внутри throwOne().");
    throw new IllegalAccessException("демонстрация");
  }

  public static void main(String[] args) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Перехвачено " + e);
    }
  }
}
```

Программа выдает следующий вывод:

```
Внутри throwOne().
Перехвачено java.lang.IllegalAccessException: демонстрация
```

Конструкция finally

Когда генерируются исключения, поток выполнения в методе направляется по довольно резкому нелинейному пути, изменяющем нормальный ход выполнения метода. В зависимости от того, как закодирован метод, исключение может даже привести к преждевременному возврату из метода, что в некоторых методах может стать проблемой. Например, если метод открывает файл при входе и закрывает его при выходе, то пропуск кода, закрывающего файл, механизмом обработки исключений нельзя считать приемлемым. Для такой нештатной ситуации и предназначено ключевое слово `finally`.

Ключевое слово `finally` позволяет создать блок кода, который будет выполняться после завершения блока `try/catch` и перед кодом, следующим после `try/catch`. Блок `finally` будет выполняться независимо от того, сгенерировано исключение или нет. В случае генерации исключения блок `finally` будет выполняться, даже если исключение не соответствует ни одной конструкции `catch`. Каждый раз, когда метод собирается вернуть управление вызывающему коду из блока `try/catch` через неперехваченное исключение или явно посредством оператора `return`, блок `finally` тоже выполняется непосредственно перед возвратом из метода. Таким образом, с помощью блока `finally` удобно закрывать файловые дескрипторы и освобождать любые другие ресурсы, которые могли быть выделены в начале метода с намерением освобождения их перед возвратом. Конструкция `finally` является необязательной. Тем не менее, для каждого оператора `try` требуется хотя бы одна конструкция `catch` или `finally`.

Ниже приведен пример программы с тремя методами, которые завершаются разными способами, не пропуская выполнение конструкции `finally`:

```
// Демонстрация применения finally.
class FinallyDemo {
  // Сгенерировать исключение внутри метода.
  static void procA() {
    try {
      System.out.println("Внутри метода procA()");
      throw new RuntimeException("демонстрация");
    } finally {
      System.out.println("Блок finally метода procA()");
    }
  }
  // Возвратить управление изнутри блока try.
  static void procB() {
    try {
      System.out.println("Внутри метода procB()");
      return;
    } finally {
      System.out.println("Блок finally метода procB()");
    }
  }
  // Выполнить блок try обычным образом.
  static void procC() {
```

```
    try {
      System.out.println("Внутри метода procC()");
    } finally {
      System.out.println("Блок finally метода procC()");
    }
  }
  public static void main(String[] args) {
    try {
      procA();
    } catch (Exception e) {
      System.out.println("Исключение перехвачено");
    }
    procB();
    procC();
  }
}
```

Оператор `try` в методе `procA()` преждевременно прерывается генерацией исключения. Блок `finally` выполняется при выходе. Оператор `try` в методе `procB()` завершается оператором `return`. Блок `finally` выполняется до возврата из `procB()`. В методе `procC()` оператор `try` выполняется нормально, без ошибок. Однако блок `finally` все равно выполняется.

Помните! Если с оператором `try` ассоциирован блок `finally`, то этот блок будет выполнен по завершении `try`.

Вот вывод, полученный в результате запуска предыдущей программы:

```
Внутри метода procA()
Блок finally метода procA()
Исключение перехвачено
Внутри метода procB()
Блок finally метода procB()
Внутри метода procC()
Блок finally метода procC()
```

Встроенные исключения Java

Внутри стандартного пакета `java.lang` определено несколько классов исключений Java. Некоторые из них использовались в предшествующих примерах. Наиболее общие из них являются подклассами стандартного типа `RuntimeException`. Как объяснялось ранее, такие исключения не нужно включать в список `throws` любого метода. На языке Java они называются *непроверяемыми исключениями*, потому что компилятор не проверяет, обрабатывает метод подобные исключения или же генерирует их. Непроверяемые исключения, определенные в `java.lang`, описаны в табл. 10.1. В табл. 10.2 перечислены те исключения, определенные в `java.lang`, которые должны помещаться в список `throws` метода, если метод может генерировать одно из исключений и не обрабатывает его самостоятельно. Они называются *проверяемыми исключениями*. Помимо исключений из `java.lang` в Java определено еще несколько, которые относятся к другим стандартным пакетам.

Таблица 10.1. Подклассы `RuntimeException` непроверяемых исключений Java, определенные в `java.lang`

Исключение	Описание
`ArithmeticException`	Арифметическая ошибка, такая как деление на ноль
`ArrayIndexOutOfBounds Exception`	Выход за допустимые пределы индекса в массиве
`ArrayStoreException`	Присваивание элементу массива значения несовместимого типа
`ClassCastException`	Недопустимое приведение
`EnumConstant NotPresentException`	Попытка использования неопределенного значения перечисления
`IllegalArgumentException`	Использование недопустимого аргумента при вызове метода
`IllegalCallerException`	Метод не может быть законно выполнен вызывающим кодом
`IllegalMonitorState Exception`	Недопустимая операция монитора, такая как ожидание неблокированного потока
`IllegalStateException`	Некорректное состояние среды или приложения
`IllegalThreadStateException`	Несовместимость запрошенной операции с текущим состоянием потока
`IndexOutOfBoundsException`	Выход за допустимые пределы индекса некоторого вида
`LayerInstantiationException`	Невозможность создания уровня модуля
`NegativeArraySizeException`	Создание массива с отрицательным размером
`NullPointerException`	Недопустимое использование ссылки `null`
`NumberFormatException`	Недопустимое преобразование строки в числовой формат
`SecurityException`	Попытка нарушения безопасности
`StringIndexOutOfBounds Exception`	Попытка индексации за границами строки
`TypeNotPresentException`	Тип не найден
`UnsupportedOperation Exception`	Обнаружение неподдерживаемой операции

Таблица 10.2. Классы проверяемых исключений Java, определенные в `java.lang`

Исключение	Описание
`ClassNotFoundException`	Класс не найден
`CloneNotSupportedException`	Попытка клонирования объекта, который не реализует интерфейс `Cloneable`
`IllegalAccessException`	Доступ к классу запрещен
`InstantiationException`	Попытка создания объекта абстрактного класса или интерфейса
`InterruptedException`	Один поток был прерван другим потоком
`NoSuchFieldException`	Запрошенное поле не существует
`NoSuchMethodException`	Запрошенный метод не существует
`ReflectiveOperationException`	Суперкласс исключений, связанных с рефлексией

Создание собственных подклассов `Exception`

Хотя встроенные исключения Java обрабатывают наиболее распространенные ошибки, вполне вероятно, что вы захотите создать собственные типы исключений, которые подходят для ситуаций, специфичных для ваших приложений. Делается это довольно легко: нужно просто определить подкласс `Exception` (который, конечно же, является подклассом `Throwable`). Вашим подклассам фактически ничего не придется реализовывать — одно их существование в системе типов позволяет использовать их как исключения. В самом классе `Exception` никаких методов не определено. Разумеется, он наследует методы, предоставляемые `Throwable`. Таким образом, все исключения, в том числе созданные вами, имеют доступные для них методы, которые определены в классе `Throwable` и описаны в табл. 10.3. Вы также можете переопределить один или несколько из этих методов в создаваемых классах исключений.

Таблица 10.3. Методы, определенные в классе `Throwable`

Метод	Описание
`final void addSuppressed(Throwable exc)`	Добавляет `exc` в список подавляемых исключений, ассоциированный с вызывающим исключением. Метод предназначен главным образом для использования в операторе `try` с ресурсами
`Throwable fillInStackTrace()`	Возвращает объект `Throwable`, который содержит полную трассировку стека. Этот объект может быть сгенерирован повторно

Окончание табл. 10.3

Метод	Описание
`Throwable getCause()`	Возвращает исключение, которое лежит в основе текущего исключения. Если лежащее в основе исключение отсутствует, тогда возвращается `null`
`String getLocalizedMessage()`	Возвращает локализованное описание исключения
`String getMessage()`	Возвращает описание исключения
`StackTraceElement[] getStackTrace()`	Возвращает массив объектов `StackTraceElement`, содержащий поэлементную трассировку стека. Метод на верхушке стека — это тот, который был вызван последним перед генерацией исключения. Он находится в первом элементе массива. Класс `StackTraceElement` предоставляет программе доступ к информации о каждом элементе в трассировке стека, такой как имя метода
`final Throwable[] getSuppressed()`	Получает подавляемые исключения, ассоциированные с вызываемым исключением, и возвращает массив, который содержит результат. Подавляемые исключения генерируются главным образом оператором `try` с ресурсами
`Throwable initCause(Throwable causeExc)`	Связывает `causeExc` с вызывающим исключением как причину его возникновения. Возвращает ссылку на исключение
`void printStackTrace()`	Отображает трассировку стека
`void printStackTrace(PrintStream stream)`	Посылает трассировку стека в указанный поток
`void printStackTrace(PrintWriter stream)`	Посылает трассировку стека в указанный поток
`void setStackTrace(StackTraceElement[] elements)`	Устанавливает трассировку стека в элементы, переданные в параметре `elements`. Предназначен для специализированного, а не нормального применения
`String toString()`	Возвращает объект `String`, содержащий описание исключения. Вызывается оператором `println()` при выводе объекта `Throwable`

В классе Exception определены четыре открытых конструктора. Два из них поддерживают сцепленные исключения, обсуждаемые в следующем разделе, а другие два показаны ниже:

```
Exception()
Exception(String msg)
```

Первая форма конструктора создает исключение, не имеющее описания. Вторая форма конструктора позволяет указать в msg описание исключения.

Хотя указание описания при создании исключения часто полезно, иногда лучше переопределить метод toString() и вот почему: версия toString(), определенная в классе Throwable (и унаследованная классом Exception), сначала отображает имя исключения, за которым следует двоеточие и ваше описание. Переопределив toString(), вы можете запретить отображение имени исключения и двоеточия, сделав вывод более чистым, что желательно в некоторых случаях.

В следующем примере объявляется новый подкласс Exception, который затем используется для сигнализации об ошибке в методе. В подклассе переопределяется метод toString(), позволяя отобразить аккуратно настроенное описание исключения.

```
// В этой программе создается специальный тип исключения.
class MyException extends Exception {
  private int detail;

  MyException(int a) {
    detail = a;
  }

  public String toString() {
    return "MyException[" + detail + "]";
  }
}

class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Вызов compute(" + a + ")");
    if(a > 10)
      throw new MyException(a);
    System.out.println("Нормальное завершение");
  }

  public static void main(String[] args) {
    try {
      compute(1);
      compute(20);
    } catch (MyException e) {
      System.out.println("Перехвачено исключение " + e);
    }
  }
}
```

В примере определяется подкласс Exception по имени MyException. Он довольно прост: имеет только конструктор и переопределенный ме-

тод toString(), который отображает значение исключения. В классе ExceptionDemo определен метод calculate(), который создает объект MyException. Исключение генерируется, когда значение целочисленного параметра calculate() больше 10. В методе main() устанавливается обработчик исключений для MyException, после чего вызывается метод calculate() с допустимым значением (меньше 10) и недопустимым, чтобы продемонстрировать оба пути через код. Вот результат:

```
Вызов compute(1)
Нормальное завершение
Вызов compute(20)
Перехвачено исключение MyException[20]
```

Сцепленные исключения

Несколько лет назад в подсистему исключений было включено средство под названием *сцепленные исключения*. Сцепленные исключения позволяют ассоциировать с одним исключением другое исключение, которое описывает причину первого исключения. Например, представьте ситуацию, в которой метод выдает исключение ArithmeticException из-за попытки деления на ноль. Тем не менее, фактической причиной проблемы было возникновение ошибки ввода-вывода, из-за которой делитель установился неправильно. Хотя метод, безусловно, должен генерировать исключение ArithmeticException, т.к. произошла именно указанная ошибка, вы можете сообщить вызывающему коду, что основной причиной была ошибка ввода-вывода. Сцепленные исключения позволяют справиться с этой и любой другой ситуацией, в которой существуют уровни исключений.

Чтобы сделать возможными сцепленные исключения, в класс Throwable были добавлены два конструктора и два метода. Конструкторы приведены ниже:

```
Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)
```

В первой форме causeExc является исключением, которое привело к возникновению текущего исключения, т.е. представляет собой основную причину его возникновения. Вторая форма позволяет указать описание одновременно с указанием причины исключения. Эти два конструктора также были добавлены в классы Error, Exception и RuntimeException.

Класс Throwable поддерживает методы для сцепленных исключений — getCause() и initCause(), которые были описаны в табл. 10.3 и повторяются здесь ради целей обсуждения:

```
Throwable getCause()
Throwable initCause(Throwable causeExc)
```

Метод getCause() возвращает исключение, лежащее в основе текущего исключения. Если лежащего в основе исключения нет, тогда возвращается null. Метод initCause() связывает causeExc с вызывающим исключением

и возвращает ссылку на исключение. Таким образом, вы можете ассоциировать причину с исключением после его создания. Однако исключение-причина может быть установлено только один раз, т.е. вызывать `initCause()` для каждого объекта исключения допускается только один раз. Кроме того, если исключение-причина было установлено конструктором, то вы не можете установить его снова с помощью `initCause()`. В общем, метод `initCause()` используется с целью установки причины для устаревших классов исключений, которые не поддерживают два дополнительных конструктора, описанных ранее.

Ниже приведен пример, иллюстрирующий механику обработки сцепленных исключений:

```
// Демонстрация работы сцепленных исключений.
class ChainExcDemo {
  static void demoproc() {

    // Создать исключение.
    NullPointerException e =
      new NullPointerException("верхний уровень");

    // Добавить причину.
    e.initCause(new ArithmeticException("причина"));

    throw e;
  }

  public static void main(String[] args) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      // Отобразить исключение верхнего уровня.
      System.out.println("Перехвачено: " + e);

      // Отобразить исключение-причину.
      System.out.println("Первоначальная причина: " + e.getCause());
    }
  }
}
```

Вот вывод, полученный в результате запуска программы:

```
Перехвачено: java.lang.NullPointerException: верхний уровень
Первоначальная причина: java.lang.ArithmeticException: причина
```

В приведенном примере исключением верхнего уровня является `NullPointerException`. К нему добавлено исключение-причина, `ArithmeticException`. Когда исключение генерируется из `demoproc()`, оно перехватывается в `main()`. Там отображается исключение верхнего уровня, за которым следует базовое исключение, получаемое путем вызова `getCause()`.

Сцепленные исключения могут быть реализованы с любой необходимой глубиной. Таким образом, исключение-причина может и само иметь причину. Имейте в виду, что слишком длинные цепочки исключений могут указывать на неудачное проектное решение.

Сцепленные исключения — это не то, что нужно каждой программе. Тем не менее, в тех случаях, когда полезно знать основную причину, они предлагают элегантное решение.

Три дополнительных средства в системе исключений

В версии JDK 7 к системе исключений были добавлены три интересных и полезных средства. Первое средство автоматизирует процесс освобождения ресурса, такого как файл, когда он больше не нужен. Оно основано на расширенной форме оператора `try` под названием `try` с ресурсами и описано в главе 13, где обсуждаются файлы. Второе средство называется множественным *перехватом*, а третье иногда упоминается как *финальная повторная генерация* или *более точная повторная генерация*. Последние два средства описаны ниже.

Средство множественного перехвата позволяет перехватывать два или более исключений одной и той же конструкцией `catch`. Нередко два или более обработчика исключений применяют ту же самую кодовую последовательность, даже если реагируют на разные исключения. Вместо перехвата каждого типа исключения по отдельности можно использовать одну конструкцию `catch` для обработки всех исключений без дублирования кода.

Для применения множественного перехвата необходимо объединить все типы исключений в конструкции `catch` с помощью операции "ИЛИ". Каждый параметр множественного перехвата неявно является `final`. (При желании `final` можно указывать явно, но это не обязательно.) Поскольку каждый параметр множественного перехвата неявно является `final`, ему нельзя присваивать новое значение.

Вот конструкция `catch`, которая использует средство множественного перехвата для `ArithmeticException` и `ArrayIndexOutOfBoundsException`:

```
catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
```

Ниже средство множественного перехвата демонстрируется в действии:

```
// Демонстрация средства множественного перехвата.
class MultiCatch {
  public static void main(String[] args) {
    int a=10, b=0;
    int[] vals = { 1, 2, 3 };
    try {
      int result = a / b;  // сгенерировать исключение ArithmeticException
      // vals[10] = 19;    // сгенерировать исключение
                           //   ArrayIndexOutOfBoundsException
      // Следующая конструкция catch перехватывает оба исключения.
    } catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
      System.out.println("Перехвачено исключение: " + e);
    }
    System.out.println("После множественного перехвата.");
  }
}
```

Программа сгенерирует исключение `ArithmeticException` при попытке деления на ноль. Если вы закомментируете оператор с делением и удалите символ комментария в следующей строке, тогда будет создано исключение `ArrayIndexOutOfBoundsException`. Оба исключения перехватываются одной конструкцией `catch`.

Средство более точной повторной генерации ограничивает тип исключений, которые могут повторно генерироваться, только теми проверяемыми исключениями, которые выдает связанный блок `try`, которые не обрабатываются предыдущей конструкцией `catch` и которые являются подтипом или супертипом параметра. Хотя такая возможность может требоваться нечасто, теперь она доступна для использования. Чтобы задействовать средство более точной повторной генерации, параметр `catch` обязан быть либо фактически `final`, т.е. ему не должно присваиваться новое значение внутри блока `catch`, либо явным образом объявляться как `final`.

Использование исключений

Обработка исключений предоставляет мощный механизм управления сложными программами, обладающими множеством динамических характеристик во время выполнения. Важно думать о `try`, `throw` и `catch` как об аккуратных способах обработки ошибок и необычных граничных условий в логике вашей программы. Вместо применения кодов возврата для обозначения ошибок используйте возможности Java по обработке исключений. Таким образом, когда метод может отказать, он должен генерировать исключение. Это более ясный способ обработки режимов отказа.

И последнее замечание: операторы обработки исключений Java не следует рассматривать как общий механизм нелокального ветвления, потому что это только запутает код и затруднит его сопровождение.

ГЛАВА 11 Многопоточное программирование

В языке Java обеспечивается встроенная поддержка *многопоточного программирования*. Многопоточная программа состоит из двух или более частей, которые могут выполняться одновременно. Каждая часть такой программы называется *потоком*, и каждый поток определяет отдельный путь выполнения. Таким образом, многопоточность представляет собой специализированную форму многозадачности.

Вы почти наверняка знакомы с многозадачностью, потому что она поддерживается практически всеми современными операционными системами (ОС). Однако есть два разных типа многозадачности: на основе процессов и на основе потоков. Важно понимать разницу между ними. Многим читателям больше знакома многозадачность, основанная на процессах. По сути, *процесс* — это программа, которая выполняется. Таким образом, многозадачность *на основе процессов* является функциональным средством, которое позволяет вашему компьютеру запускать две или большее число программ параллельно. Например, многозадачность на основе процессов позволяет запускать компилятор Java одновременно с использованием текстового редактора или посещением веб-сайта. В многозадачности, основанной на процессах, программа представляет собой наименьшую единицу кода, которая может координироваться планировщиком.

В многозадачной среде, *основанной на потоках*, поток является наименьшей единицей координируемого кода, т.е. одна программа может выполнять две или более задач одновременно. Например, текстовый редактор может форматировать текст одновременно с его выводом на печать, если эти два действия выполняются двумя отдельными потоками. Таким образом, многозадачность на основе процессов имеет дело с "общей картиной", а многозадачность на основе потоков обрабатывает детали.

Многозадачные потоки требуют меньше накладных расходов, чем многозадачные процессы. Процессы — это тяжеловесные задачи, требующие отдельного адресного пространства. Взаимодействие между процессами является затратным и ограниченным. Переключение контекста с одного процесса на другой также обходится дорого. С другой стороны, потоки более легковесны. Они совместно используют одно и то же адресное пространство и

один и тот же тяжеловесный процесс. Взаимодействие между потоками не сопряжено с высокими затратами, а переключение контекста с одного потока на другой обходится дешевле. В то время как программы на Java применяют многозадачные среды, основанные на процессах, многозадачность на основе процессов не находится под непосредственным контролем Java. Тем не менее, многопоточная многозадачность есть.

Многопоточность позволяет писать эффективные программы, которые максимально задействуют вычислительную мощность, доступную в системе. Одним из важных способов достижения такой цели с помощью многопоточности является сведение к минимуму времени простоя, что особенно важно для интерактивной сетевой среды, в которой работает Java, поскольку время простоя будет обычным явлением. Например, скорость передачи данных по сети намного ниже скорости, с которой компьютер способен их обрабатывать. Даже ресурсы локальной файловой системы читаются и записываются гораздо медленнее, чем они могут быть обработаны процессором. И, конечно же, пользовательский ввод намного медленнее, нежели компьютер. В однопоточной среде программа должна ожидать завершения каждой задачи, прежде чем она сможет перейти к следующей задаче, даже если большую часть времени программа бездействует в ожидании ввода. Многопоточность помогает сократить это время простоя, т.к. другой поток может выполняться, когда один ожидает.

Если вы программировали для таких ОС, как Windows, то уже знакомы с многопоточным программированием. Однако тот факт, что Java управляет потоками, делает многопоточность особенно удобной, потому что многие детали выполняются за вас.

Потоковая модель Java

Исполняющая среда Java во многом зависит от потоков, и все библиотеки классов разработаны с учетом многопоточности. На самом деле Java использует потоки, чтобы вся среда была асинхронной. Это помогает снизить неэффективность за счет предотвращения потери циклов центрального процессора (ЦП).

Ценность многопоточной среды лучше всего понимается по контрасту с ее аналогом. В однопоточных системах применяется подход, называемый *циклом обработки событий* с *опросом*. В такой модели один поток управления работает в бесконечном цикле, опрашивая одну очередь событий, чтобы решить, что делать дальше. Как только этот механизм опроса возвращает, скажем, сигнал о том, что сетевой файл готов к чтению, цикл обработки событий передает управление соответствующему обработчику событий. Пока данный обработчик события не возвратит управление, в программе больше ничего не может произойти, из-за чего тратится процессорное время. Кроме того, может получиться так, что одна часть программы будет доминировать над системой и препятствовать обработке любых других событий. В целом в однопоточной среде, когда поток *блокируется* (т.е. приостанавливает выполнение) по причине ожидания некоторого ресурса, то прекращает выполнение вся программа.

Преимущество многопоточности в Java заключается в том, что механизм главного цикла/опроса устранен. Один поток может делать паузу, не останавливая другие части программы. Например, время простоя, возникающее при чтении потоком данных из сети или ожидании пользовательского ввода, можно задействовать в другом месте. Многопоточность позволяет циклам анимации приостанавливаться на секунду между каждым кадром, не вызывая паузы в работе всей системы. Когда поток блокируется в программе на Java, приостанавливается только один заблокированный поток. Все остальные потоки продолжают функционировать.

Как известно большинству читателей, за последние несколько лет многоядерные системы стали обычным явлением. Конечно, одноядерные системы по-прежнему широко применяются. Важно понимать, что многопоточность Java работает в системах обоих типов. В одноядерной системе одновременно выполняющиеся потоки совместно используют ЦП, при этом каждый поток получает долю процессорного времени. Таким образом, в одноядерной системе два или более потока фактически не выполняются одновременно, но задействуется время простоя ЦП. Однако в многоядерных системах возможно одновременное выполнение двух или более потоков. Во многих случаях удается еще больше повысить эффективность программы и увеличить скорость выполнения определенных операций.

На заметку! В дополнение к функциям многопоточности, описанным в этой главе, вас наверняка заинтересует инфраструктура Fork/Join Framework. Она предлагает мощные средства для создания многопоточных приложений, которые автоматически масштабируются с целью наилучшего использования многоядерных сред. Инфраструктура Fork/Join Framework является частью поддержки *параллельного программирования* в Java, которое представляет собой обычное название методик, оптимизирующих некоторые типы алгоритмов для параллельного выполнения в системах с несколькими процессорами. Обсуждение Fork/Join Framework и других утилит параллелизма ищите в главе 29. Здесь описаны традиционные возможности многопоточности Java.

Потоки пребывают в нескольких состояниях. Далее приведено общее описание. Поток может *выполняться*. Он может быть *готовым к запуску*, как только получит время ЦП. Работающий поток может быть *приостановлен*, в результате чего он временно прекращает свою активность. Выполнение приостановленного потока затем можно *возобновить*, позволяя ему продолжиться с того места, где он остановился. Поток может быть *заблокирован* при ожидании ресурса. В любой момент работа потока может быть *прекращена*, что немедленно останавливает его выполнение. После прекращения работы выполнение потока не может быть возобновлено.

Приоритеты потоков

Каждому потоку в Java назначается приоритет, который определяет способ обработки данного потока в сравнении с другими потоками. Приоритеты потоков представляют собой целые числа, определяющие относительный

приоритет одного потока над другим. Как абсолютная величина приоритет не имеет смысла; поток с более высоким приоритетом не работает быстрее потока с более низким приоритетом, если он является единственным функционирующим потоком. Взамен приоритет потока применяется для принятия решения о том, когда переключаться с одного работающего потока на другой. Прием называется *переключением контекста*. Правила, которые определяют, когда происходит переключение контекста, просты и описаны ниже.

- *Поток может добровольно передать управление.* Это происходит при явной уступке очереди, приостановке или блокировке. В таком сценарии проверяются все остальные потоки, и поток с наивысшим приоритетом, готовый к выполнению, получает ЦП.
- *Поток может быть вытеснен потоком с более высоким приоритетом.* В таком случае поток с более низким приоритетом, который не уступает ЦП, просто вытесняется — независимо от того, что он делает — потоком с более высоким приоритетом. По сути, как только поток с более высоким приоритетом желает запуститься, он это делает. Прием называется *вытесняющей многозадачностью*.

В случаях, когда два потока с одинаковым приоритетом конкурируют за циклы ЦП, ситуация немного усложняется. В некоторых ОС потоки с одинаковым приоритетом автоматически разделяются по времени в циклическом режиме. Для других типов ОС потоки с равным приоритетом должны добровольно передавать управление своим партнерам. Если они этого не сделают, тогда другие потоки не запустятся.

Внимание! Проблемы с переносимостью могут возникать из-за различий в том, как ОС переключают контекст для потоков с одинаковым приоритетом.

Синхронизация

Поскольку многопоточность привносит асинхронное поведение в ваши программы, должен быть способ обеспечения синхронности, когда она необходима. Скажем, если вы хотите, чтобы два потока взаимодействовали и совместно использовали сложную структуру данных, такую как связный список, то вам нужен способ предотвращения конфликтов между ними. То есть вы должны запретить одному потоку записывать данные, пока другой поток находится в процессе их чтения. Для этой цели в Java реализован элегантный вариант старой модели синхронизации между процессами: *монитор*. Монитор представляет собой механизм управления, который был впервые определен Ч.Э.Р. Хоаром. Вы можете думать о мониторе как об очень маленьком "ящике", способном содержать только один поток. Как только поток входит в монитор, все остальные потоки должны ждать, пока этот поток не выйдет из монитора. Таким образом, монитор можно применять для защиты общего ресурса от манипулирования более чем одним потоком одновременно.

В Java нет класса, который назывался бы "Monitor"; взамен каждый объект имеет собственный неявный монитор, в который автоматически осуществляется вход при вызове одного из синхронизированных методов объекта. Как только поток оказывается внутри синхронизированного метода, остальные потоки не могут вызывать какой-либо другой синхронизированный метод для того же объекта. Это позволяет писать очень ясный и лаконичный многопоточный код, поскольку в язык встроена поддержка синхронизации.

Обмен сообщениями

После разделения своей программы на отдельные потоки вам нужно определить, как они будут взаимодействовать друг с другом. Для установления связи между потоками в ряде других языков вы должны полагаться на ОС. Разумеется, накладные расходы возрастают. Напротив, Java предоставляет нескольким потокам ясный и экономичный способ взаимодействия друг с другом через вызовы предопределенных методов, которые есть у всех объектов. Система обмена сообщениями Java позволяет потоку войти в синхронизированный метод объекта и затем ожидать, пока какой-то другой поток явно не уведомит его о выходе.

Класс `Thread` и интерфейс `Runnable`

Многопоточная система Java построена на основе классе `Thread`, его методах и дополняющем интерфейсе `Runnable`. Класс `Thread` инкапсулирует поток выполнения. Поскольку вы не можете напрямую ссылаться на нематериальное состояние работающего потока, то будете иметь с ним дело через его посредника, т.е. работать с экземпляром `Thread`, который его породил. Чтобы создать новый поток, ваша программа либо расширит класс `Thread`, либо реализует интерфейс `Runnable`.

В классе `Thread` определено несколько методов, помогающих управлять потоками. Некоторые из тех, что использовались в этой главе, перечислены в табл. 11.1.

Таблица 11.1. Методы, помогающие управлять потоками

Метод	Описание
`getName()`	Получает имя потока
`getPriority()`	Получает приоритет потока
`isAlive()`	Определяет, выполняется ли поток
`join()`	Ожидает прекращения работы потока
`run()`	Устанавливает точку входа в поток
`sleep()`	Приостанавливает выполнение потока на указанное время
`start()`	Запускает поток вызовом его метода `run()`

До сих пор во всех примерах, приведенных в книге, применялся один поток выполнения. Далее в главе будет объясняться, как использовать Thread и Runnable для создания потоков и управления ими, начиная с одного потока, который есть во всех программах Java: главного потока.

Главный поток

Когда программа Java запускается, немедленно начинает выполняться один поток. Обычно его называют *главным потоком* программы, потому что именно он выполняется при ее запуске. Главный поток важен по двум причинам:

- он является потоком, из которого будут порождаться другие "дочерние" потоки;
- часто он должен заканчивать выполнение последним, поскольку выполняет разнообразные действия по завершению работы.

Хотя главный поток создается автоматически при запуске программы, им можно управлять с помощью объекта Thread. Для этого понадобится получить ссылку на него, вызвав метод currentThread(), который является открытым статическим членом класса Thread со следующим общим видом:

```
static Thread currentThread()
```

Этот метод возвращает ссылку на поток, в котором он вызывается. Когда у вас есть ссылка на главный поток, вы можете управлять им так же, как и любым другим потоком.

Начнем с рассмотрения показанного ниже примера:

```
// Управление главным потоком.
class CurrentThreadDemo {
  public static void main(String[] args) {
    Thread t = Thread.currentThread();
    System.out.println("Текущий поток: " + t);
    // Изменить имя потока.
    t.setName("My Thread");
    System.out.println("После изменения имени: " + t);
    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван");
    }
  }
}
```

В приведенной программе ссылка на текущий (в данном случае главный) поток получается вызовом currentThread() и сохраняется в локальной переменной t. Затем программа отображает информацию о потоке, вызывает setName() для изменения внутреннего имени потока и повторно отображает информацию о потоке. Далее выполняется обратный отсчет от пяти до

единицы с паузой, составляющей одну секунду, между выводом строк. Пауза достигается методом sleep(). Аргумент функции sleep() задает период задержки в миллисекундах. Обратите внимание на блок try/catch, внутрь которого помещен цикл. Метод sleep() в Thread может сгенерировать исключение InterruptedException в случае, если какой-то другой поток пожелает прервать этот спящий поток. В данном примере просто вводится сообщение, если поток прерывается, но в реальной программе пришлось бы поступать по-другому. Вот вывод, выдаваемый программой:

```
Текущий поток: Thread[main,5,main]
После изменения имени: Thread[My Thread,5,main]
5
4
3
2
1
```

Обратите внимание на результат, когда t используется в качестве аргумента функции println(). В результате по порядку отображается имя потока, его приоритет и имя его группы. По умолчанию именем главного потока является main. Его приоритет равен 5 — стандартному значению, и main также будет именем группы потоков, к которой принадлежит текущий поток. *Группа потоков* представляет собой структуру данных, которая управляет состоянием набора потоков в целом. После изменения имени потока переменная t снова выводится. На этот раз отображается новое имя потока.

Давайте более подробно рассмотрим методы класса Thread, которые применяются в программе. Метод sleep() заставляет поток, из которого он вызывается, приостанавливать выполнение на заданный период времени в миллисекундах. Ниже показан его общий вид:

```
static void sleep(long milliseconds) throws InterruptedException
```

Количество миллисекунд для приостановки указывается в аргументе milliseconds. Метод sleep() может сгенерировать исключение InterruptedException.

Метод sleep() имеет вторую форму, которая позволяет указывать период времени в миллисекундах и наносекундах:

```
static void sleep(long milliseconds, int nanoseconds) throws InterruptedException
```

Вторая форма sleep() полезна только в средах, которые допускают временные периоды с точностью до наносекунд.

В предыдущей программе видно, что с помощью setName() вы можете устанавливать имя потока. Получить имя потока можно, вызвав getName() (но обратите внимание, что в программе это не показано). Упомянутые методы являются членами класса Thread и объявлены следующим образом:

```
final void setName(String threadName)
final String getName()
```

В threadName указывается имя потока.

Создание потока

В самом общем случае поток создается путем создания экземпляр объекта типа Thread. В языке Java предусмотрены два способа:

- можно реализовать интерфейс Runnable;
- можно расширить класс Thread.

В последующих двух разделах рассматриваются оба способа.

Реализация интерфейса Runnable

Самый простой способ создать поток предусматривает создание класса, реализующего интерфейс Runnable, который абстрагирует единицу исполняемого кода. Вы можете создать поток для любого объекта, реализующего Runnable. Для реализации Runnable в классе понадобится реализовать только один метод с именем run(), объявление которого показано ниже:

```
public void run()
```

Внутрь метода run() помещается код, который основывает новый поток. Важно понимать, что run() может вызывать другие методы, использовать другие классы и объявлять переменные в точности, как это делает главный поток. Единственное отличие заключается в том, что метод run() устанавливает точку входа для другого параллельного потока выполнения в программе. Этот поток завершится, когда управление возвратится из run().

После создания класса, реализующего интерфейс Runnable, внутри него создается объект типа Thread. В классе Thread определено несколько конструкторов. Вот тот, который будет применяться:

```
Thread(Runnable threadOb, String threadName)
```

В приведенном конструкторе threadOb является экземпляром класса, реализующего интерфейс Runnable. Он определяет, где начнется выполнение потока. Имя нового потока определяется параметром threadName.

После создания новый поток не будет запущен, пока не вызовется его метод start(), объявленный в классе Thread. По существу start() инициирует вызов run(). Метод start() показан ниже:

```
void start()
```

Далее приведен пример создания нового потока и запуска его выполнения:

```
// Создание второго потока.
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Создать новый, второй поток.
    t = new Thread(this, "Demo Thread");
    System.out.println("Дочерний поток: " + t);
  }

  // Это точка входа для второго потока.
```

```
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Дочерний поток: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Дочерний поток прерван.");
    }
    System.out.println("Завершение дочернего потока.");
  }
}
class ThreadDemo {
  public static void main(String[] args) {
    NewThread nt = new NewThread();    // создать новый поток
    nt.t.start();                      // запустить поток
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Главный поток: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван.");
    }
    System.out.println("Завершение главного потока.");
  }
}
```

Новый объект `Thread` создается внутри конструктора класса `NewThread` с помощью следующего оператора:

```
t = new Thread(this, "Demo Thread");
```

Передача `this` в качестве первого аргумента указывает на то, что новый поток должен вызвать метод `run()` для данного объекта. Внутри `main()` вызывается `start()`, который запускает поток выполнения, начиная с метода `run()`. Это приводит к тому, что цикл `for` дочернего потока начинает свою работу. Затем главный поток входит в цикл `for`. Оба потока продолжают работать, совместно используя ЦП в одноядерных системах, пока их цикл не завершится. Результат работы программы представлен ниже. (Выходные данные могут отличаться в зависимости от конкретной среды выполнения.)

```
Дочерний поток: Thread[Demo Thread,5,main]
Главный поток: 5
Дочерний поток: 5
Дочерний поток: 4
Главный поток: 4
Дочерний поток: 3
Дочерний поток: 2
Главный поток: 3
Дочерний поток: 1
Завершение дочернего потока.
Главный поток: 2
Главный поток: 1
Завершение главного потока.
```

Как упоминалось ранее, в многопоточной программе часто бывает полезно, чтобы главный поток заканчивал работу последним. Предыдущая программа гарантирует, что главный поток завершится последним, потому что он засыпает на 1000 миллисекунд между итерациями, а дочерний поток — только на 500 миллисекунд. В итоге дочерний поток завершается раньше, чем главный. Вскоре вы увидите лучший способ дождаться завершения потока.

Расширение класса Thread

Второй способ создания потока предусматривает создание нового класса, расширяющего Thread, и создание экземпляра этого класса. Расширяющий класс должен переопределить метод run(), который является точкой входа для нового потока. Как и раньше, вызов start() начинает выполнение нового потока. Вот предыдущая программа, переписанная для расширения Thread:

```
// Создание второго потока путем расширения класса Thread.
class NewThread extends Thread {

  NewThread() {
    // Создать новый, второй поток.
    super("Demo Thread");
    System.out.println("Дочерний поток: " + this);
  }

  // Это точка входа для второго потока.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Дочерний поток: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Дочерний поток прерван.");
    }
    System.out.println("Завершение дочернего потока.");
  }
}

class ExtendThread {
  public static void main(String[] args) {
    NewThread nt = new NewThread();     // создать новый поток
    nt.start();                         // запустить поток
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Главный поток: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван.");
    }
    System.out.println("Завершение главного потока.");
  }
}
```

Программа генерирует тот же самый результат, что и предыдущая версия. Как видите, дочерний поток создается путем создания объекта класса NewThread, производного от Thread.

Обратите внимание на вызов super() внутри конструктора NewThread, который приводит к вызову следующей формы конструктора Thread:

```
public Thread(String threadName)
```

В аргументе threadName указывается имя потока.

Выбор подхода

На данном этапе вам может быть интересно, почему в Java есть два способа создания дочерних потоков и какой подход лучше. Ответы на упомянутые вопросы касаются одного и того же. В классе Thread имеется несколько методов, которые могут быть переопределены в производном классе. Единственным методом, который *должен* быть переопределен, является run(). Конечно, это тот же самый метод, требующийся при реализации интерфейса Runnable. Многие программисты на Java считают, что классы следует расширять только тогда, когда они каким-то образом совершенствуются или адаптируются. Таким образом, если вы не собираетесь переопределять какие-то другие методы Thread, то вероятно лучше просто реализовать Runnable. Кроме того, благодаря реализации Runnable ваш класс потока не придется наследовать от Thread, что позволяет наследовать его от другого класса. В конечном итоге только вам решать, какой подход использовать. Тем не менее, в оставшемся материале главы потоки будут создаваться с применением классов, реализующих интерфейс Runnable.

Создание множества потоков

До сих пор использовались только два потока: главный поток и один дочерний поток. Однако в программе можно создавать столько потоков, сколько необходимо. Например, в следующей программе создаются три дочерних потока:

```
// Создание множества потоков.
class NewThread implements Runnable {
  String name;  // имя потока
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("Новый поток: " + t);
  }

  // Это точка входа для потока.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
```

```
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " прерван");
    }
    System.out.println(name + " завершен.");
  }
}

class MultiThreadDemo {
  public static void main(String[] args) {
    NewThread nt1 = new NewThread("One");
    NewThread nt2 = new NewThread("Two");
    NewThread nt3 = new NewThread("Three");

    // Запустить потоки.
    nt1.t.start();
    nt2.t.start();
    nt3.t.start();

    try {
      // Ожидать окончания остальных потоков.
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван");
    }

    System.out.println("Завершение главного потока.");
  }
}
```

Ниже показан пример вывода, генерируемого программой. (Ваш вывод может отличаться в зависимости от конкретной среды выполнения.)

```
Новый поток: Thread[One,5,main]
Новый поток: Thread[Two,5,main]
Новый поток: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One завершен.
Two завершен.
Three завершен.
Завершение главного потока.
```

Как видите, после запуска все три дочерних потока совместно используют ЦП. Обратите внимание на вызов `sleep(10000)` в `main()`, который приводит к засыпанию главного потока на период в десять секунд и гарантирует, что он завершится последним.

Использование `isAlive()` и `join()`

Как уже упоминалось, часто требуется, чтобы главный поток заканчивался последним. В предшествующих примерах это достигается вызовом `sleep()` в `main()` с достаточно большой задержкой, что гарантирует завершение всех дочерних потоков до главного потока. Тем не менее, вряд ли такое решение можно считать удовлетворительным, и оно также поднимает более важный вопрос: как один поток может узнать, когда другой поток закончился? К счастью, класс `Thread` предлагает средства, с помощью которых можно ответить на данный вопрос.

Существуют два способа выяснения, завершен ли поток. Во-первых, можно вызвать на потоке метод `isAlive()`, который определен в классе `Thread` со следующей общей формой:

```
final boolean isAlive()
```

Метод `isAlive()` возвращает `true`, если поток, на котором он был вызван, все еще выполняется, или `false` в противном случае.

Хотя метод `isAlive()` иногда бывает полезным, для ожидания завершения потока вы чаще всего будете применять метод по имени `join()`:

```
final void join() throws InterruptedException
```

Этот метод ожидает завершения потока, на котором он вызывается. Его имя происходит от концепции вызывающего потока, ожидающего до тех пор, пока указанный поток не *присоединится* к нему. Дополнительные формы метода `join()` позволяют указывать максимальное время ожидания завершения указанного потока.

Вот улучшенная версия предыдущего примера, где метод `join()` используется для того, чтобы основной поток останавливался последним. Вдобавок демонстрируется применение метода `isAlive()`.

```
// Использование join() для ожидания окончания потоков.
class NewThread implements Runnable {
  String name;  // имя потока
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("Новый поток: " + t);
  }

  // Это точка входа для потока.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
```

```
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " прерван.");
    }
    System.out.println(name + " завершен.");
  }
}

class DemoJoin {
  public static void main(String[] args) {
    NewThread nt1 = new NewThread("One");
    NewThread nt2 = new NewThread("Two");
    NewThread nt3 = new NewThread("Three");

    // Запустить потоки.
    nt1.t.start();
    nt2.t.start();
    nt3.t.start();

    System.out.println("Поток One работает: "
                       + nt1.t.isAlive());
    System.out.println("Поток Two работает: "
                       + nt2.t.isAlive());
    System.out.println("Поток Three работает: "
                       + nt3.t.isAlive());
    // Ожидать завершения потоков.
    try {
      System.out.println("Ожидание завершения потоков.");
      nt1.t.join();
      nt2.t.join();
      nt3.t.join();
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван");
    }

    System.out.println("Поток One работает: "
                       + nt1.t.isAlive());
    System.out.println("Поток Two работает: "
                       + nt2.t.isAlive());
    System.out.println("Поток Three работает: "
                       + nt3.t.isAlive());

    System.out.println("Завершение главного потока.");
  }
}
```

Ниже показан пример вывода, генерируемого программой. (Ваш вывод может отличаться в зависимости от конкретной среды выполнения.)

```
Новый поток: Thread[One,5,main]
Новый поток: Thread[Two,5,main]
Новый поток: Thread[Three,5,main]
Поток One работает: true
Поток Two работает: true
Поток Three работает: true
```

```
Ожидание завершения потоков.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two завершен.
Three завершен.
One завершен.
Поток One работает: false
Поток Two работает: false
Поток Three работает: false
Завершение главного потока.
```

Как видите, после возврата управления из вызовов `join()` потоки остановили свое выполнение.

Приоритеты потоков

Приоритеты потоков используются планировщиком потоков для принятия решения о том, когда следует разрешить выполнение каждого потока. Теоретически в течение заданного периода времени потоки с более высоким приоритетом получают больше времени ЦП, чем потоки с более низким приоритетом. На практике количество времени ЦП, которое получает поток, часто зависит от нескольких факторов помимо его приоритета. (Например, способ реализации многозадачности в ОС может повлиять на относительную доступность времени ЦП.) Поток с более высоким приоритетом может также вытеснять поток с более низким приоритетом. Скажем, когда выполняется поток с более низким приоритетом и возобновляется выполнение потока с более высоким приоритетом (например, происходит его выход из режима сна или ожидания ввода-вывода), он вытесняет поток с более низким приоритетом.

Теоретически потоки с одинаковыми приоритетами должны иметь равный доступ к процессору. Но вам нужно проявлять осторожность. Помните, что язык Java предназначен для работы в широком диапазоне сред. Некоторые из таких сред реализуют многозадачность принципиально иначе, чем другие. В целях безопасности потоки с одинаковыми приоритетами должны время от времени уступать управление. Подход гарантирует, что все потоки смогут работать в ОС без вытеснения. На практике даже в средах без вытеснения

большинство потоков по-прежнему имеют возможность запускаться, поскольку большинство потоков неизбежно сталкиваются с некоторыми блокирующими ситуациями, такими как ожидание ввода-вывода. Когда возникает подобная ситуация, заблокированный поток приостанавливается, и другие потоки могут выполняться. Но если вы хотите плавного многопоточного выполнения, тогда вам лучше не полагаться на это. Кроме того, некоторые типы задач сильно загружают ЦП. Потоки такого рода всецело поглощают ЦП. Для потоков этих типов понадобится время от времени передавать управление, чтобы могли выполняться другие потоки.

Для установки приоритета потока применяется метод `setPriority()`, который является членом класса `Thread`. Вот его общая форма:

```
final void setPriority(int level)
```

В аргументе `level` указывается новая настройка приоритета для вызывающего потока. Значение `level` должно находиться в диапазоне от `MIN_PRIORITY` до `MAX_PRIORITY`. В настоящее время эти значения равны `1` и `10` соответственно. Чтобы вернуть потоку стандартный приоритет, необходимо указать значение `NORM_PRIORITY`, которое в настоящее время равно `5`. Упомянутые приоритеты определены как статические финальные переменные внутри `Thread`.

Получить текущую настройку приоритета можно вызовом метода `getPriority()` класса `Thread`:

```
final int getPriority()
```

Что касается диспетчеризации потоков, то реализации Java могут вести себя совершенно по-разному. Большинство несоответствий возникает при наличии потоков, которые полагаются на вытесняющее поведение вместо того, чтобы совместно уступать ЦП. Самый безопасный способ добиться предсказуемого межплатформенного поведения Java предусматривает использование потоков, которые добровольно уступают управление ЦП.

Синхронизация

Когда двум или большему числу потоков требуется доступ к общему ресурсу, нужно каким-нибудь способом гарантировать, что ресурс будет эксплуатироваться только одним потоком в каждый момент времени. Процесс, с помощью которого достигается такая цель, называется *синхронизацией*. Вы увидите, что поддержка синхронизации в Java обеспечивается на уровне языка.

Ключом к синхронизации является концепция монитора. *Монитор* представляет собой объект, который применяется в качестве взаимоисключающей блокировки. В заданный момент времени *владеть* монитором может только один поток. Когда поток получает блокировку, то говорят, что он *входит* в монитор. Все другие потоки, пытающиеся войти в заблокированный монитор, будут приостановлены до тех пор, пока первый поток не *выйдет* из монитора. Говорят, что эти другие потоки *ожидают* монитор. Поток, владеющий монитором, может при желании повторно войти в тот же самый монитор.

Синхронизировать код можно одним из двух способов, предполагающих использование ключевого слова synchronized. Оба способа рассматриваются далее.

Использование синхронизированных методов

Синхронизацию в Java обеспечить легко, потому что все объекты имеют собственные связанные с ними неявные мониторы. Чтобы войти в монитор объекта, понадобится лишь вызвать метод, модифицированный с помощью ключевого слова synchronized. Пока поток находится внутри синхронизированного метода, все другие потоки, пытающиеся вызвать его (или любой другой синхронизированный метод) на том же экземпляре, должны ожидать. Чтобы выйти из монитора и передать управление объектом следующему ожидающему потоку, владелец монитора просто возвращает управление из синхронизированного метода.

Для понимания потребности в синхронизации давайте начнем с несложного примера, в котором она не применяется, хотя и должна. В следующей программе определены три простых класса. Первый, Callme(), имеет единственный метод call(), принимающий строковый параметр по имени msg. Данный метод пытается вывести строку msg внутри квадратных скобок. Интересно отметить, что в методе call() после вывода открывающей скобки и строки msg производится вызов Thread.sleep(1000), который приостанавливает текущий поток на одну секунду.

Конструктор показанного ниже класса Caller получает в аргументах target и msg ссылку на экземпляр класса Callme и строку. Конструктор также создает новый поток, который будет вызывать метод run() этого объекта. Метод run() класса Caller вызывает метод call() экземпляра target класса Callme, передавая строку msg. Наконец, класс Synch начинает с создания одного экземпляра Callme и трех экземпляров Caller, каждый с уникальной строкой сообщения. Каждому экземпляру Caller передается тот же самый экземпляр Callme.

```
// Эта программа не синхронизирована.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch(InterruptedException e) {
      System.out.println("Прерван");
    }
    System.out.println("]");
  }
}

class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
```

```
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
  }
  public void run() {
    target.call(msg);
  }
}
class Synch {
  public static void main(String[] args) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");
    // Запустить потоки.
    ob1.t.start();
    ob2.t.start();
    ob3.t.start();
    // Ожидать окончания работы потоков.
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Прерван");
    }
  }
}
```

Вот вывод, генерируемый программой:

```
[Hello[Synchronized[World]
]
]
```

Как видите, за счет вызова `sleep()` метод `call()` позволяет переключиться на другой поток, что приводит к смешанному выводу трех строк сообщений. В программе нет ничего такого, что помешало бы всем трем потокам одновременно вызывать один и тот же метод для того же самого объекта. Ситуация известна как *состояние гонок*, поскольку три потока соперничают друг с другом за завершение метода. В приведенном примере используется метод `sleep()`, чтобы сделать эффекты повторяемыми и очевидными. В большинстве ситуаций состояние гонок является более тонким и менее предсказуемым, т.к. трудно предугадать, когда произойдет переключение контекста. В итоге программа один раз может выполниться правильно, а другой — неправильно.

Для исправления предыдущей программы потребуется *сериализировать* доступ к `call()`, т.е. ограничить его доступ только к одному потоку за раз. Для этого просто нужно указать перед определением `call()` ключевое слово `synchronized`:

```
class Callme {
  synchronized void call(String msg) {
    ...
```

Ключевое слово synchronized не позволяет остальным потокам входить в метод call(), пока он используется другим потоком. После добавления synchronized к определению call() вывод будет следующим:

```
[Hello]
[Synchronized]
[World]
```

При наличии метода или группы методов, которые манипулируют внутренним состоянием объекта в многопоточном сценарии, должно применяться ключевое слово synchronized, чтобы защитить состояние от условий гонок. Помните, что как только поток входит в какой-либо синхронизированный метод в экземпляре, никакой другой поток не может входить в любой другой синхронизированный метод в том же экземпляре. Однако несинхронизированные методы этого экземпляра по-прежнему доступны для вызова.

Оператор synchronized

Несмотря на то что определение синхронизированных методов внутри создаваемых классов является простым и эффективным средством обеспечения синхронизации, оно не будет работать во всех случаях. Чтобы понять причину, рассмотрим следующую ситуацию. Представьте, что вы хотите синхронизировать доступ к объектам класса, не предназначенного для многопоточного доступа, т.е. синхронизированные методы в классе не используются. Кроме того, данный класс создан не вами, а третьей стороной, и у вас нет доступа к исходному коду. Таким образом, вы не можете добавить ключевое слово synchronized к соответствующим методам внутри класса. Как синхронизировать доступ к объекту такого класса? К счастью, решить проблему довольно легко: вы просто помещаете вызовы методов, определенных этим классом, внутрь блока synchronized.

Ниже показана общая форма оператора synchronized:

```
synchronized(objRef) {
  // операторы, подлежащие синхронизации
}
```

Здесь objRef — это ссылка на синхронизируемый объект. Блок synchronized гарантирует, что вызов синхронизированного метода, который является членом класса objRef, произойдет только после того, как текущий поток успешно войдет в монитор objRef.

Далее представлена альтернативная версия предыдущего примера, где внутри метода run() применяется блок synchronized:

```
// В этой программе используется блок synchronized.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
```

```
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Прерван");
    }
    System.out.println("]");
  }
}
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;

  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
  }

  // Синхронизированные вызовы call().
  public void run() {
    synchronized(target) {  // блок synchronized
      target.call(msg);
    }
  }
}
class Synch1 {
  public static void main(String[] args) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

    // Запустить потоки.
    ob1.t.start();
    ob2.t.start();
    ob3.t.start();

    // Ожидать окончания работы потоков.
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Прерван");
    }
  }
}
```

Метод `call()` не модифицируется с помощью `synchronized`. Взамен внутри метода `run()` класса `Caller` используется оператор `synchronized`, что приводит к такому же корректному выводу, как и в предыдущем примере, поскольку каждый поток ожидает завершения работы предыдущего потока, прежде чем продолжить свое выполнение.

Взаимодействие между потоками

В предшествующих примерах выполнялась безусловная блокировка асинхронного доступа к определенным методам для других потоков. Такое применение неявных мониторов в объектах Java дает мощный эффект, но вы можете достичь более тонкого уровня контроля посредством взаимодействия между процессами. Как вы увидите, делать это в Java особенно легко.

Как обсуждалось ранее, многопоточность заменяет программирование с циклом событий, разделяя ваши задачи на дискретные логические единицы. Потоки также обеспечивают дополнительное преимущество: они устраняю опрос. Опрос обычно реализуется в виде цикла, который используется для повторяющейся проверки некоторого условия. Как только условие становится истинным, инициируется соответствующее действие, что приводит к ненужному расходу времени ЦП. Например, рассмотрим классическую задачу организации очереди, когда один поток производит некоторые данные, а другой поток их потребляет. Чтобы сделать задачу более интересной, предположим, что производителю нужно дождаться завершения работы потребителя, прежде чем он сгенерирует дополнительные данные. В системе опроса потребитель будет тратить много циклов ЦП в ожидании, пока производитель начнет генерацию. После того как производитель закончил работу, потребитель начинает опрос, понапрасну тратя еще больше циклов ЦП на ожидание завершения работы потребителя, и т.д. Ясно, что ситуация подобного рода нежелательна.

В Java имеется элегантный механизм взаимодействия между процессами с помощью методов `wait()`, `notify()` и `notifyAll()`, который позволяет избежать опроса. Указанные методы реализованы как финальные в классе `Object`, поэтому они есть во всех классах. Все три метода можно вызывать только из синхронизированного контекста. Хотя они концептуально сложны с вычислительной точки зрения, правила их применения в действительности довольно просты.

- Метод `wait()` сообщает вызывающему потоку о необходимости уступить монитор и перейти в спящий режим, пока какой-то другой поток не войдет в тот же монитор и не вызовет `notify()` или `notifyAll()`.
- Метод `notify()` пробуждает поток, который вызвал `wait()` на том же самом объекте.
- Метод `notifyAll()` пробуждает все потоки, которые вызвали `wait()` на том же самом объекте. Одному из этих потоков будет предоставлен доступ.

Методы объявлены в классе `Object`, как показано ниже:

```
final void wait() throws InterruptedException
final void notify()
final void notifyAll()
```

Существуют дополнительные формы метода wait(), которые позволяют указывать период времени для ожидания.

Прежде чем приступить к рассмотрению примера, иллюстрирующего взаимодействие между потоками, необходимо сделать одно важное замечание. Хотя метод wait() обычно ожидает, пока не будет вызван метод notify() или notifyAll(), существует вероятность того, что в очень редких случаях ожидающий поток может быть разбужен из-за *ложного пробуждения*. В этом случае ожидающий поток возобновляется без вызовов notify() или notifyAll(). (В сущности, поток возобновляется без видимой причины.) Из-за такой маловероятной возможности в документации по Java API рекомендуется, чтобы вызовы wait() выполнялись в цикле, проверяющем условие ожидания потока. Прием демонстрируется в следующем примере.

А теперь рассмотрим пример, в котором используются методы wait() и notify(). Для начала рассмотрим приведенный далее пример программы, в которой неправильно реализована простая форма задачи с производителем и потребителем. Реализация состоит из четырех классов: Q — очередь, которую вы пытаетесь синхронизировать; Producer — потоковый объект, создающий записи в очереди; Consumer — потоковый объект, потребляющий записи очереди; PC — крошечный класс, который создает объекты Q, Producer и Consumer.

```
// Некорректная реализация производителя и потребителя.
class Q {
  int n;

  synchronized int get() {
    System.out.println("Получено: " + n);
    return n;
  }

  synchronized void put(int n) {
    this.n = n;
    System.out.println("Отправлено: " + n);
  }
}

class Producer implements Runnable {
  Q q;
  Thread t;

  Producer(Q q) {
    this.q = q;
    t = new Thread(this, "Производитель");
  }

  public void run() {
    int i = 0;

    while(true) {
      q.put(i++);
    }
  }
}
```

```
class Consumer implements Runnable {
  Q q;
  Thread t;

  Consumer(Q q) {
    this.q = q;
    t = new Thread(this, "Потребитель");
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}

class PC {
  public static void main(String[] args) {
    Q q = new Q();
    Producer p = new Producer(q);
    Consumer c = new Consumer(q);

    // Запустить потоки.
    p.t.start();
    c.t.start();

    System.out.println("Нажмите <Ctrl-C>, чтобы остановить программу.");
  }
}
```

Несмотря на то что методы `put()` и `get()` в классе `Q` синхронизированы, ничто не помешает производителю переполнить потребителя, равно как ничто не мешает потребителю дважды использовать одно и то же значение из очереди. Таким образом, вы получите ошибочный вывод, показанный ниже (точный вывод будет зависеть от быстродействия ЦП и рабочей нагрузки):

```
Отправлено: 1
Получено: 1
Получено: 1
Получено: 1
Получено: 1
Получено: 1
Отправлено: 2
Отправлено: 3
Отправлено: 4
Отправлено: 5
Отправлено: 6
Отправлено: 7
Получено: 7
```

Как видите, после отправки производителем значения 1 потребитель запустился и получил одно и то же значение 1 пять раз подряд. Затем производитель возобновил работу и отправил значения от 2 до 7, не дав потребителю возможности их получить.

Чтобы корректно написать эту программу на Java, необходимо применять методы `wait()` и `notify()` для передачи сигналов в обоих направлениях:

```
// Корректная реализация производителя и потребителя.
class Q {
  int n;
  boolean valueSet = false;

  synchronized int get() {
    while(!valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("Перехвачено исключение InterruptedException");
      }

      System.out.println("Получено: " + n);
      valueSet = false;
      notify();
      return n;
  }

  synchronized void put(int n) {
    while(valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("Перехвачено исключение InterruptedException");
      }

      this.n = n;
      valueSet = true;
      System.out.println("Отправлено: " + n); notify();
  }
}

class Producer implements Runnable {
  Q q;
  Thread t;

  Producer(Q q) {
    this.q = q;
    t = new Thread(this, "Производитель");
  }

  public void run() {
    int i = 0;
    while(true) {
      q.put(i++);
    }
  }
}

class Consumer implements Runnable {
  Q q;
  Thread t;

  Consumer(Q q) {
    this.q = q;
    t = new Thread(this, "Потребитель");
  }
```

```
  public void run() {
    while(true) {
      q.get();
    }
  }
}
class PCFixed {
  public static void main(String[] args) {
    Q q = new Q();
    Producer p = new Producer(q);
    Consumer c = new Consumer(q);

    // Запустить потоки.
    p.t.start();
    c.t.start();

    System.out.println("Нажмите <Ctrl-C>, чтобы остановить программу.");
  }
}
```

Внутри метода `get()` вызывается `wait()`, обеспечивая приостановку его выполнения до тех пор, пока объект `Producer` не уведомит вас о готовности данных. Когда это произойдет, выполнение внутри `get()` возобновится. После получения данных метод `get()` вызывает `notify()`, сообщая объекту `Producer` о том, что в очередь можно поместить дополнительные данные. Внутри метода `put()` функция `wait()` приостанавливает выполнение до тех пор, пока объект `Consumer` не удалит элемент из очереди. Когда выполнение возобновляется, следующий элемент данных помещается в очередь и вызывается `notify()`, указывая объекту на то, что теперь он должен удалить его.

Вот часть вывода, генерируемого программой, который наглядно демонстрирует чистое синхронное поведение:

```
Отправлено: 1
Получено: 1
Отправлено: 2
Получено: 2
Отправлено: 3
Получено: 3
Отправлено: 4
Получено: 4
Отправлено: 5
Получено: 5
```

Взаимоблокировка

Важно избегать особого вида ошибок, связанного именно с многозадачностью и называемого *взаимоблокировкой*, которая возникает в ситуации, когда два потока имеют циклическую зависимость от пары синхронизированных объектов. Например, пусть один поток входит в монитор объекта X, а другой поток — в монитор объекта Y. Если поток в X попытается вызвать какой-то синхронизированный метод на объекте Y, то он вполне ожидаемо заблокируется. Тем не менее, если поток в Y в свою очередь попытается вызвать какой-

либо синхронизированный метод на объекте X, то он будет ожидать вечно, т.к. для доступа к X ему пришлось бы освободить собственную блокировку на Y, чтобы первый поток мог завершиться. Взаимоблокировка — трудная для отладки ошибка по двум причинам.

- В целом взаимоблокировка случается очень редко, если два потока правильно распределяют время.
- Она способна вовлечь более двух потоков и двух синхронизированных объектов. (То есть взаимоблокировка может возникнуть из-за более запутанной последовательности событий, чем только что описанная.)

Для более полного понимания взаимоблокировки полезно взглянуть на нее в действии. В следующем примере создаются два класса, A и B, с методами `foo()` и `bar()` соответственно, которые ненадолго приостанавливаются перед попыткой вызвать метод другого класса. Главный класс по имени `Deadlock` создает экземпляры A и B, после чего вызывает метод `deadlockStart()`, чтобы запустить второй поток, который настраивает условие взаимоблокировки. Методы `foo()` и `bar()` используют `sleep()` как способ вызвать состояние взаимоблокировки.

```
// Пример возникновения взаимоблокировки.
class A {
  synchronized void foo(B b) {
    String name = Thread.currentThread().getName();
    System.out.println(name + " вошел в A.foo");

    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("A прерван");
    }

    System.out.println(name + " пытается вызвать B.last()");
    b.last();
  }

  synchronized void last() {
    System.out.println("Внутри A.last()");
  }
}

class B {
  synchronized void bar(A a) {
    String name = Thread.currentThread().getName();
    System.out.println(name + " вошел в B.bar");

    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("B прерван");
    }

    System.out.println(name + " пытается вызвать A.last()");
    a.last();
  }
```

```
    synchronized void last() {
      System.out.println("Внутри B.last()");
    }
  }
  class Deadlock implements Runnable {
    A a = new A();
    B b = new B();
    Thread t;
    Deadlock() {
      Thread.currentThread().setName("MainThread");
      t = new Thread(this, "RacingThread");
    }
    void deadlockStart() {
      t.start();
      a.foo(b);  // получить блокировку на a в этом потоке
      System.out.println("Назад в главный поток");
    }
    public void run() {
      b.bar(a);  // получить блокировку на b в другом потоке
      System.out.println("Назад в другой поток");
    }
    public static void main(String[] args) {
      Deadlock dl = new Deadlock();
      dl.deadlockStart();
    }
  }
```

Запустив программу, вы увидите приведенный далее вывод, хотя какой метод будет запущен первым — `A.foo()` или `B.bar()`, — зависит от конкретной среды выполнения.

```
MainThread вошел в A.foo
RacingThread вошел в B.bar
MainThread пытается вызвать B.last()
RacingThread пытается вызвать A.last()
```

Поскольку в программе произошла взаимоблокировка, для завершения программы понадобится нажать <Ctrl+C>. Нажав комбинацию клавиш <Ctrl+Break> на ПК, можно просмотреть полный дамп кеша потока и монитора. Вы увидите, что `RacingThread` владеет монитором на `b`, пока он ожидает монитор на `a`. В то же время `MainThread` владеет `a` и ожидает получения `b`. Программа никогда не завершится. Как иллюстрирует данный пример, если ваша многопоточная программа иногда зависает, то взаимоблокировка является одним из первых условий, которые вы должны проверить.

Приостановка, возобновление и останов потоков

Временами полезно приостанавливать выполнение потока. Например, отдельный поток можно применять для отображения времени суток. Если пользователю не нужны часы, тогда его поток можно приостановить. В любом случае приостановить поток довольно просто. После приостановки перезапустить поток тоже легко.

Механизмы приостановки, останова и возобновления потоков различаются между ранними версиями Java, такими как Java 1.0, и более современными версиями, начиная с Java 2. До выхода Java 2 в программе использовались методы `suspend()`, `resume()` и `stop()`, которые определены в классе `Thread` и предназначены для приостановки, возобновления и останова выполнения потока. Хотя упомянутые методы кажутся разумным и удобным подходом к управлению выполнением потоков, они не должны применяться в новых программах на Java и вот почему. Несколько лет назад в версии Java 2 метод `suspend()` класса `Thread` был объявлен нерекомендуемым. Так поступили из-за того, что `suspend()` мог иногда становиться причиной серьезных системных отказов. Предположим, что поток заблокировал критически важные структуры данных. Если этот поток приостановить в данной точке, то блокировки не освободятся. Другие потоки, ожидающие такие ресурсы, могут попасть во взаимоблокировку.

Метод `resume()` тоже не рекомендуется использовать. Проблем он не вызывает, но его нельзя применять без дополняющего метода `suspend()`.

Метод `stop()` класса `Thread` тоже объявлен нерекомендуемым в Java 2. Причина в том, что иногда он мог приводить к серьезным системным отказам. Например, пусть поток выполняет запись в критически важную структуру данных и завершил внесение только части нужных изменений. Если такой поток будет остановлен в этот момент, тогда структура данных может остаться в поврежденном состоянии. Дело в том, что метод `stop()` вызывает освобождение любой блокировки, удерживаемой вызывающим потоком. Таким образом, поврежденные данные могут быть использованы другим потоком, ожидающим той же блокировки.

Поскольку применять методы `suspend()`, `resume()` или `stop()` для управления потоком теперь нельзя, вам может показаться, что больше не существует какого-либо способа для приостановки, перезапуска или завершения работы потока. К счастью, это не так. Взамен поток должен быть спроектирован таким образом, чтобы метод `run()` периодически проверял, должен ли поток приостанавливать, возобновлять или полностью останавливать собственное выполнение. Как правило, задача решается путем установления флаговой переменной, которая указывает состояние выполнения потока. Пока флаговая переменная установлена в состояние "работает", метод `run()` должен продолжать работу, чтобы позволить потоку выполняться. Если флаговая переменная установлена в состояние "приостановить", то поток должен быть приостановлен. Если она установлена в состояние "остановить", тогда поток должен завершить работу. Конечно, существует множество способов написания такого кода, но главный принцип будет одинаковым во всех программах.

```
// Приостановка и возобновление современным способом.
class NewThread implements Runnable {
  String name;  // имя потока
  Thread t;
  boolean suspendFlag;
  NewThread(String threadname) {
    name = threadname;
```

```
    t = new Thread(this, name);
    System.out.println("Новый поток: " + t);
    suspendFlag = false;
  }
  // Это точка входа для потока.
  public void run() {
    try {
      for(int i = 15; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(200);
        synchronized(this) {
          while(suspendFlag) {
            wait();
          }
        }
      }
    } catch (InterruptedException e) {
      System.out.println(name + " прерван.");
    }
    System.out.println(name + " завершается.");
  }
  synchronized void mysuspend() {
    suspendFlag = true;
  }
  synchronized void myresume() {
    suspendFlag = false;
    notify();
  }
}
class SuspendResume {
  public static void main(String[] args) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    ob1.t.start();  // запустить поток
    ob2.t.start();  // запустить поток
    try {
      Thread.sleep(1000);
      ob1.mysuspend();
      System.out.println("Приостановка потока One");
      Thread.sleep(1000);
      ob1.myresume();
      System.out.println("Возобновление потока One");
      ob2.mysuspend();
      System.out.println("Приостановка потока Two");
      Thread.sleep(1000);
      ob2.myresume();
      System.out.println("Возобновление потока Two");
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван");
    }
    // Ожидать завершения потоков.
    try {
      System.out.println("Ожидание завершения потоков.");
      ob1.t.join();
      ob2.t.join();
```

```
        } catch (InterruptedException e) {
          System.out.println("Главный поток прерван");
        }
        System.out.println("Главный поток завершается.");
      }
    }
```

Запустив программу, вы заметите, что потоки приостанавливают и возобновляют свою работу. Далее в книге вы увидите больше примеров, где используется современный механизм управления потоками. Хотя этот механизм может показаться не настолько простым в применении, как старый, однако, он необходим для предотвращения возникновения ошибок во время выполнения. Он является подходом, который *должен* использоваться в любом новом коде.

Получение состояния потока

Как упоминалось ранее в главе, поток может пребывать в разных состояниях. Получить текущее состояние потока можно за счет вызова метода `getState()`, определенного в классе `Thread`:

```
Thread.State getState()
```

Метод `getState()` возвращает значение типа `Thread.State`, отражающее состояние потока на момент выполнения вызова. Состояние представлено в виде перечисления, определенного в `Thread`. (Перечисление — это список именованных констант и подробно обсуждается в главе 12.) В табл. 11.2. описаны значения, которые может возвращать метод `getState()`.

Таблица 11.2. Значения, возвращаемые методом `getState()`

Значение	Состояние
`BLOCKED`	Поток приостановил выполнение, потому что ожидает получения блокировки
`NEW`	Поток еще не начал выполнение
`RUNNABLE`	Поток либо в текущий момент выполняется, либо будет выполняться, когда получит доступ к ЦП
`TERMINATED`	Поток завершил выполнение
`TIMED_WAITING`	Поток приостановил выполнение на указанный период времени, например, при вызове `sleep()`. Поток переходит в это состояние также в случае вызова версии `wait()` или `join()`, принимающей значение тайм-аута
`WAITING`	Поток приостановил выполнение из-за того, что ожидает возникновения некоторого действия. Например, он находится в состоянии `WAITING` по причине вызова версии `wait()` или `join()`, не принимающей значение тайм-аута

На рис. 11.1 показаны взаимосвязи между различными состояниями потока.

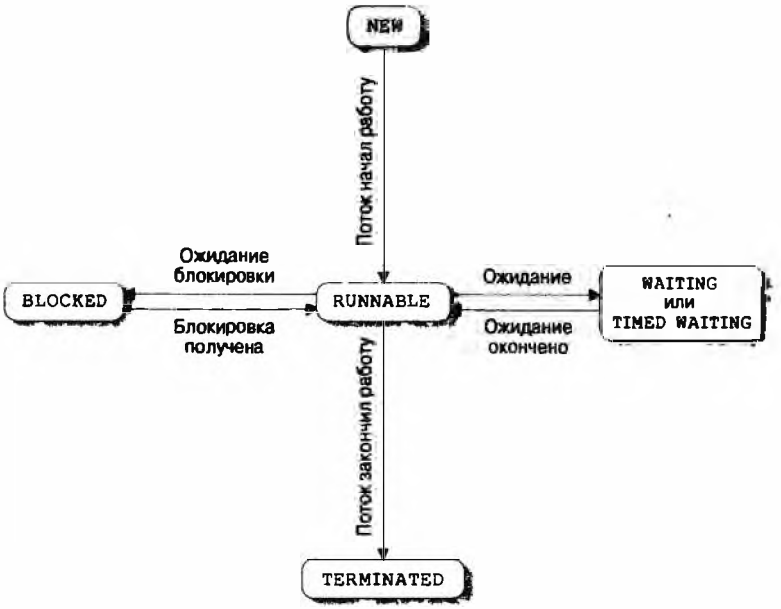


Рис. 11.1. Состояния потока

Имея экземпляр Thread, вы можете применить метод `getState()` для получения состояния потока. Скажем, в следующей кодовой последовательности выясняется, пребывает ли поток по имени `thrd` в состоянии RUNNABLE в момент вызова `getState()`:

```
Thread.State ts = thrd.getState();
if(ts == Thread.State.RUNNABLE)  // ...
```

Важно понимать, что после вызова `getState()` состояние потока может измениться. Таким образом, в зависимости от обстоятельств состояние, полученное при вызове метода `getState()`, спустя всего лишь мгновение может уже не отражать фактическое состояние потока. По этой и другим причинам метод `getState()` не предназначен быть средством синхронизации потоков. В основном он используется для отладки или профилирования характеристик потока во время выполнения.

Использование фабричных методов для создания и запуска потока

В ряде случаев нет необходимости отделять создание потока от запуска его на выполнение. Другими словами, иногда удобно создать и запустить поток одновременно. Один из способов сделать предусматривает применение статического фабричного метода. *Фабричный метод* возвращает объект класса. Как правило, фабричные методы являются *статическими* методами класса.

Они используются по разным причинам, например, для установки объекта в начальное состояние перед работой с ним, для настройки определенного типа объекта или временами для обеспечения многократного использования объекта. Что касается создания и запуска потока, то фабричный метод создаст поток, вызовет метод `start()` на потоке и возвратит ссылку на поток. При таком подходе вы можете создавать и запускать поток с помощью одного вызова метода, тем самым упрощая свой код. Скажем, добавление в класс `NewThread` из приведенной в начале главы программы `ThreadDemo` показанного далее фабричного метода позволит создавать и запускать поток за один шаг:

```
// Фабричный метод, который создает и запускает поток.
public static NewThread createAndStart() {
  NewThread myThrd = new NewThread();
  myThrd.t.start();
  return myThrd;
}
```

С применением метода `createAndStart()` следующий фрагмент кода:

```
NewThread nt = new NewThread();   // создать новый поток
nt.t.start();                     // запустить поток
```

можно заменить так:

```
NewThread nt = NewThread.createAndStart();
```

Теперь поток создается и запускается за один шаг.

В ситуациях, когда хранить ссылку на исполняемый поток не нужно, иногда можно создать и запустить поток с помощью одной строки кода без использования фабричного метода. Снова вернувшись к программе `ThreadDemo`, вот как создать и запустить поток `NewThread`:

```
new NewThread().t.start();
```

Тем не менее, в реальных приложениях обычно требуется сохранять ссылку на поток, поэтому фабричный метод часто будет удачным вариантом.

Использование многопоточности

Ключом к эффективному применению возможностей многопоточности Java является мышление в категориях параллельного, а не последовательного выполнения. Скажем, если у вас есть две подсистемы в программе, которые могут выполняться одновременно, сделайте их отдельными потоками. При осторожном использовании многопоточности вы можете создавать очень эффективные программы. Однако имейте в виду, что если вы создадите слишком много потоков, то на самом деле можете ухудшить производительность своей программы, а не улучшить ее. Помните, что с переключением контекста связаны некоторые накладные расходы. Если вы создадите слишком много потоков, то большая часть времени ЦП будет расходоваться на изменение контекстов, а не на выполнение самой программы! И последнее замечание: для создания приложений с интенсивными вычислениями, которые могут автоматически масштабироваться, чтобы задействовать доступные процессоры в многоядерной системе, рассмотрите возможность применения инфраструктуры Fork/Join Framework, которая описана в главе 29.

ГЛАВА

12 Перечисления, автоупаковка и аннотации

В настоящей главе рассматриваются три средства, которые изначально не были частью языка Java, но со временем каждое из них стало почти незаменимым аспектом программирования на Java: перечисления, автоупаковка и аннотации. Первоначально добавленные в JDK 5, все они являются функциональными возможностями, на которые стали полагаться программисты на Java, т.к. предлагают упрощенный подход к решению общих программных задач. В главе также обсуждаются оболочки типов Java и дано введение в рефлексию.

Перечисления

В своей простейшей форме *перечисление* представляет собой список именованных констант, который определяет новый тип данных и его допустимые значения. Таким образом, объект перечисления может содержать только значения, которые были объявлены в списке. Другие значения не допускаются. Иными словами, перечисление дает возможность явно указывать единственные значения, которые может законно иметь тип данных. Перечисления обычно используются для определения набора значений, представляющих коллекцию элементов. Например, вы можете применять перечисление для представления кодов ошибок, возникающих в результате выполнения какой-либо операции, таких как успех, отказ или ожидание, или списка состояний, в которых может пребывать устройство, например, рабочее, остановленное или приостановленное. В ранних версиях Java такие значения определялись с помощью переменных `final`, но перечисления предлагают гораздо более совершенный подход.

Хотя на первый взгляд перечисления в Java могут показаться похожими на перечисления в других языках, это сходство окажется поверхностным, поскольку в Java перечисление определяется как тип класса. За счет превращения перечислений в классы возможности перечисления значительно расширяются. Скажем, в Java перечисление может иметь конструкторы, методы и переменные экземпляра. Из-за своей мощи и гибкости перечисления широко используются во всей библиотеке Java API.

Основы перечислений

Перечисление создается с применением ключевого слова enum. Например, вот простое перечисление, в котором определен перечень различных сортов яблок:

```
// Перечисление сортов яблок.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

Идентификаторы Jonathan, GoldenDel и т.д. называются *константами перечисления*. Каждая из них неявно объявляется как открытый статический финальный член Apple. Более того, их типом является тип перечисления, в котором они объявлены, в данном случае — Apple. Таким образом, в языке Java такие константы называются *самотипизированными*, где "само-" относится к объемлющему перечислению.

После того как перечисление определено, можно создать переменную этого типа. Однако, несмотря на то, что перечисления определяют тип класса, экземпляр перечисления не создается с помощью new. Взамен переменная перечисления объявляется и используется почти так же, как переменная одного из примитивных типов. Например, ниже переменная ap объявляется как принадлежащая типу перечисления Apple:

```
Apple ap;
```

Поскольку ap относится к типу Apple, единственные значения, которые ей можно присваивать (или она способна содержать), определяются перечислением. Скажем, следующий оператор присваивает ap значение RedDel:

```
ap = Apple.RedDel;
```

Обратите внимание, что символ RedDel предваряется типом Apple.

Две константы перечисления можно сравнить на предмет равенства с применением операции отношения ==. Например, показанный далее оператор сравнивает значение в ap с константой GoldenDel:

```
if(ap == Apple.GoldenDel) // ...
```

Значение перечисления также можно использовать для управления оператором switch. Разумеется, во всех операторах case должны быть указаны константы из того же самого перечисления, что и в выражении switch. Например, приведенный ниже оператор switch совершенно допустим:

```
// Использование перечисления для управления оператором switch.
switch(ap) {
  case Jonathan:
    // ...
  case Winesap:
    // ...
```

Обратите внимание, что в операторах case имена констант перечисления применяются без уточнения с помощью имени их типа перечисления, т.е. используется Winesap, а не Apple.Winesap. Причина в том, что тип перечис-

ления в выражении switch уже неявно задает тип перечисления констант в case, так что нет никакой необходимости уточнять константы в операторах case посредством имени типа перечисления. На самом деле попытка сделать это вызовет ошибку на этапе компиляции.

При отображении константы перечисления, скажем, в операторе println(), выводится ее имя. Например, в результате выполнения следующего оператора отобразится имя Winesap:

```
System.out.println(Apple.Winesap);
```

В показанной ниже программе объединены все рассмотренные ранее фрагменты кода с целью демонстрации работы с перечислением Apple:

```
// Перечисление сортов яблок.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland

class EnumDemo {
  public static void main(String[] args)
  {
    Apple ap;
    ap = Apple.RedDel;
    // Вывести значение перечисления.
    System.out.println("Значение ap: " + ap);
    System.out.println();
    ap = Apple.GoldenDel;
    // Сравнить два значения перечисления.
    if(ap == Apple.GoldenDel)
      System.out.println("ap содержит GoldenDel.\n");
    // Использовать перечисление для управления оператором switch.
    switch(ap) {
      case Jonathan:
        System.out.println("Яблоки сорта Джонатан (Jonathan)
          имеют красный цвет.");
        break;
      case GoldenDel:
        System.out.println("Яблоки сорта Голден делишес (Golden Delicious)
          имеют желтый цвет.");
        break;
      case RedDel:
        System.out.println("Яблоки сорта Ред делишес (Red Delicious)
          имеют красный цвет.");
        break;
      case Winesap:
        System.out.println("Яблоки сорта Вайнсап (Winesap)
          имеют красный цвет.");
        break;
      case Cortland:
        System.out.println("Яблоки сорта Кортланд (Cortland)
           имеют красный цвет.");
        break;
    }
  }
}
```

Вот вывод, генерируемый программой:

```
Значение ap: RedDel

ap содержит GoldenDel.

Яблоки сорта Голден делишес (Golden Delicious) имеют желтый цвет.
```

Методы values() и valueOf()

Все перечисления автоматически содержат в себе два предопределенных метода: values() и valueOf() со следующими общими формами:

```
public static enum-type [] values()
public static enum-type valueOf(String str)
```

Метод values() возвращает массив, содержащий список констант перечисления. Метод valueOf() возвращает константу перечисления, значение которой соответствует строке, переданной в аргументе str. В обоих случаях в enum-type указывается тип данного перечисления. Скажем, для представленного ранее перечисления Apple возвращаемым типом Apple.valueOf("Winesap") будет Winesap.

В показанной далее программе иллюстрируется применение методов values() и valueOf():

```
// Использование встроенных методов перечисления.
// Перечисление сортов яблок.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
class EnumDemo2 {
  public static void main(String[] args)
  {
    Apple ap;
    System.out.println("Все константы перечисления Apple:");

    // Использовать values().
    Apple[] allapples = Apple.values();
    for(Apple a : allapples)
      System.out.println(a);
    System.out.println();
    // Использовать valueOf().
    ap = Apple.valueOf("Winesap");
    System.out.println("ap содержит " + ap);
  }
}
```

Вывод, генерируемый программой, выглядит следующим образом:

```
Все константы перечисления Apple:
Jonathan
GoldenDel
RedDel
Winesap
Cortland

ap содержит Winesap
```

Обратите внимание, что для прохода по массиву констант, полученному вызовом values(), в программе используется цикл for в стиле "for-each". В качестве примера была создана переменная allapples, которой присваивается ссылка на массив перечисления. Тем не менее, в таком шаге нет никакой необходимости, потому что цикл for можно было бы написать так, как показано ниже, устранив потребность в переменной allapples:

```
for(Apple a : Apple.values())
  System.out.println(a);
```

А теперь взгляните, как было получено значение, соответствующее имени Winesap, с помощью вызова valueOf():

```
ap = Apple.valueOf("Winesap");
```

Как уже объяснялось, метод valueOf() возвращает значение перечисления, которое ассоциировано с именем константы, представленной в виде строки.

Перечисления Java являются типами классов

Как уже упоминалось, перечисление Java относится к типу класса. Хотя вы не создаете экземпляр перечисления с помощью new, в остальном он обладает теми же возможностями, что и другие классы. Тот факт, что enum определяет класс, придает перечислению в Java необычайную силу. Например, вы можете предоставить ему конструкторы, добавить переменные экземпляра и методы и даже реализовать интерфейсы.

Важно понимать, что каждая константа перечисления является объектом своего типа перечисления. Таким образом, в случае определения конструктора для перечисления конструктор будет вызываться при создании каждой константы перечисления. Кроме того, каждая константа перечисления имеет собственную копию любых переменных экземпляра, определенных перечислением. Рассмотрим в качестве примера следующую версию перечисления Apple:

```
// Использование конструктора перечисления, переменной экземпляра и метода.
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
  private int price;  // цена яблок каждого сорта

  // Конструктор.
    Apple(int p) { price = p; }
    int getPrice() { return price; }
}
class EnumDemo3 {
  public static void main(String[] args)
  {
    Apple ap;

    // Отобразить цену яблок сорта Winesap.
    System.out.println("Яблоки сорта Winesap стоят " +
                       Apple.Winesap.getPrice() +
                      " центов.\n");
```

```
    // Отобразить все сорта яблок вместе с ценами.
    System.out.println("Цены на все сорта яблок:");
    for(Apple a : Apple.values())
      System.out.println("Яблоки сорта " + a + " стоят " +
                         a.getPrice() + " центов.");
  }
}
```

Вот вывод:

```
Яблоки сорта Winesap стоят 15 центов.

Цены на все сорта яблок:
Яблоки сорта Jonathan стоят 10 центов.
Яблоки сорта GoldenDel стоят 9 центов.
Яблоки сорта RedDel стоят 12 центов.
Яблоки сорта Winesap стоят 15 центов.
Яблоки сорта Cortland стоят 8 центов.
```

В эту версию перечисления `Apple` добавлены три компонента. Первый — переменная экземпляра `price`, которая применяется для хранения цены каждого сорта яблок. Второй — конструктор `Apple`, которому передается цена сорта яблок. Третий — метод `getPrice()`, возвращающий значение `price`.

Когда переменная `ap` объявляется в `main()`, конструктор `Apple` вызывается по одному разу для каждой заданной константы. Обратите внимание на способ указания аргументов конструктора за счет их помещения в круглые скобки после каждой константы:

```
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
```

Эти значения передаются параметру `p` конструктора `Apple()`, который затем присваивает его переменной экземпляра `price`. Опять-таки конструктор вызывается один раз для каждой константы.

Поскольку каждая константа перечисления имеет собственную копию `price`, вы можете получить цену определенного сорта яблок, вызвав метод `getPrice()`. Скажем, ниже показан вызов в методе `main()`, предназначенный для получения цены яблок сорта `Winesap`:

```
Apple.Winesap.getPrice()
```

Цены яблок всех сортов получаются путем прохода по перечислению с использованием цикла `for`. Ввиду того, что для каждой константы перечисления существует копия `price`, значение, связанное с одной константой, будет отдельным от значения, связанного с другой константой. Это мощная концепция, которая доступна только тогда, когда перечисления реализованы в виде классов, как сделано в Java.

Хотя в предыдущем примере содержится только один конструктор, перечисление может предлагать две или более перегруженных формы, как и любой другой класс. Например, следующая версия перечисления `Apple` предоставляет стандартный конструктор, который инициализирует цену значением `-1`, указывая на то, что данные о цене недоступны:

```
// Использование конструктора перечисления.
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);

  private int price;  // цена яблок каждого сорта

  // Конструктор.
  Apple(int p) { price = p; }

  // Перегруженный конструктор.
  Apple() { price = -1; }

  int getPrice() { return price; }
}
```

Обратите внимание, что в этой версии RedDel не имеет аргумента. Таким образом, вызывается стандартный конструктор и переменной price экземпляра RedDel присваивается значение -1.

К перечислениям применяются два ограничения. Во-первых, перечисление не может быть унаследовано от другого класса. Во-вторых, перечисление не может служить суперклассом, т.е. перечисление расширять нельзя. В остальном перечисление действует так же, как любой другой тип класса. Главное помнить, что каждая из констант перечисления является объектом класса, в котором она определена.

Перечисления унаследованы от Enum

Несмотря на невозможность при объявлении перечисления наследовать его от суперкласса, все перечисления автоматически унаследованы от java.lang.Enum. В этом классе определено несколько методов, доступных для использования всеми перечислениями. Класс Enum подробно описан в части II, но три его метода заслуживают обсуждения прямо сейчас.

Есть возможность получить значение, которое указывает позицию константы перечисления в списке констант. Оно называется *порядковым номером* и извлекается вызовом метода ordinal():

```
final int ordinal()
```

Метод ordinal() возвращает порядковый номер константы, на которой вызывается. Порядковые номера начинаются с нуля. Таким образом, в перечислении Apple константа Jonathan имеет порядковый номер 0, константа GoldenDel — порядковый номер 1, константа RedDel — порядковый номер 2 и т.д.

Порядковые номера двух констант одного и того же перечисления можно сравнивать с применением метода compareTo(). Он имеет следующую общую форму:

```
final int compareTo(enum-type e)
```

Здесь enum-type задает тип перечисления, а e представляет собой константу, сравниваемую с вызывающей константой. Не забывайте, что вызывающая константа и e должны относиться к одному и тому же перечислению. Если вызывающая константа имеет порядковый номер меньше e, то метод

compareTo() возвращает отрицательное значение. Если два порядковых номера совпадают, возвращается ноль. Если вызывающая константа имеет порядковый номер больше e, тогда возвращается положительное значение.

Константу перечисления можно сравнивать на предмет равенства с любым другим объектом, используя метод equals(), который переопределяет метод equals(), определенный в Object. Несмотря на то что метод equals() способен сравнивать константу перечисления с любым другим объектом, эти два объекта будут равны, только если они оба ссылаются на одну и ту же константу внутри того же самого перечисления. Простое наличие общих порядковых номеров не приведет к тому, что equals() возвратит true, если две константы принадлежат разным перечислениям.

Вдобавок помните о возможности сравнивать на предмет равенства две ссылки на перечисления с применением операции ==.

В приведенной далее программе демонстрируется использование методов ordinal(), compareTo() и equals():

```
// Демонстрация использования методов ordinal(), compareTo() и equals().

// Перечисление сортов яблок.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo4 {
  public static void main(String[] args)
  {
    Apple ap, ap2, ap3;

    // Получить все порядковые номера с применением ordinal().
    System.out.println("Все константы перечисления Apple" +
                       " вместе с их порядковыми номерами: ");
    for(Apple a : Apple.values())
      System.out.println(a + " " + a.ordinal());

    ap = Apple.RedDel;
    ap2 = Apple.GoldenDel;
    ap3 = Apple.RedDel;

    System.out.println();

    // Демонстрация использования compareTo() и equals().
    if(ap.compareTo(ap2) < 0)
      System.out.println(ap + " находится перед " + ap2);

    if(ap.compareTo(ap2) > 0)
      System.out.println(ap2 + " находится перед " + ap);

    if(ap.compareTo(ap3) == 0)
      System.out.println(ap + " равно " + ap3);

    System.out.println();

    if(ap.equals(ap2))
      System.out.println("Ошибка!");

    if(ap.equals(ap3))
      System.out.println(ap + " равно " + ap3);
```

```
    if(ap == ap3)
      System.out.println(ap + " == " + ap3);
  }
}
```

Ниже показан вывод, генерируемый программой:

```
Все константы перечисления Apple вместе с их порядковыми номерами:
Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4

GoldenDel находится перед RedDel
RedDel равно RedDel

RedDel равно RedDel
RedDel == RedDel
```

Еще один пример перечисления

Прежде чем двигаться дальше, мы рассмотрим еще один пример, в котором применяется перечисление. В главе 9 была создана автоматизированная программа для принятия решений. В той версии переменные с именами NO (Нет), YES (Да), MAYBE (Возможно), LATER (Позже), SOON (Скоро) и NEVER (Никогда) были объявлены внутри интерфейса и использовались для представления возможных ответов. Хотя формально в таком подходе нет ничего ошибочного, перечисление будет более удачным выбором. Вот улучшенная версия программы, в которой для определения ответов применяется перечисление по имени Answers. Сравните данную версию с первоначальной в главе 9.

```
// Улучшенная версия "системы принятия решений" из главы 9.
// В этой версии для представления ответов используется
// перечисление, а не переменные интерфейса.

import java.util.Random;

// Перечисление возможных ответов.
enum Answers {
  NO, YES, MAYBE, LATER, SOON, NEVER
}

class Question {
  Random rand = new Random();
  Answers ask() {
    int prob = (int) (100 * rand.nextDouble());
    if (prob < 15)
      return Answers.MAYBE;   // 15%
    else if (prob < 30)
      return Answers.NO;      // 15%
    else if (prob < 60)
      return Answers.YES;     // 30%
    else if (prob < 75)
      return Answers.LATER;   // 15%
```

```
    else if (prob < 98)
      return Answers.SOON;    // 13%
    else
      return Answers.NEVER;   // 2%
  }
}

class AskMe {
  static void answer(Answers result) {
    switch(result) {
      case NO:
        System.out.println("Нет");
        break;
      case YES:
        System.out.println("Да");
        break;
      case MAYBE:
        System.out.println("Возможно");
        break;
      case LATER:
        System.out.println("Позже");
        break;
      case SOON:
        System.out.println("Скоро");
        break;
      case NEVER:
        System.out.println("Никогда");
        break;
    }
  }
  public static void main(String[] args) {
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
  }
}
```

Оболочки типов

Как вам известно, в Java для хранения значений основных типов данных, поддерживаемых языком, используются примитивные типы (также называемые простыми типами) вроде `int` или `double`. Использование для хранения таких величин примитивных типов вместо объектов объясняется стремлением увеличить производительность. Применение объектов для хранения этих значений привело бы к добавлению неприемлемых накладных расходов даже к самым простым вычислениям. Таким образом, примитивные типы не являются частью иерархии объектов и не наследуются от `Object`.

Несмотря на преимущество в производительности, обеспечиваемое примитивными типами, бывают случаи, когда может понадобиться объектное

представление. Скажем, передавать примитивный тип по ссылке в метод нельзя. Кроме того, многие стандартные структуры данных, реализованные в Java, работают с объектами, следовательно, вы не можете использовать такие структуры данных для хранения значений примитивных типов. Чтобы справиться с этими (и другими) ситуациями, в языке Java предусмотрены *оболочки типов*, которые представляют собой классы, инкапсулирующие примитивный тип внутри объекта. Классы оболочек типов подробно описаны в части II, но здесь они представлены из-за своей прямой связи со средством автоупаковки Java.

К оболочкам типов относятся `Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`, `Character` и `Boolean`. Перечисленные классы предлагают широкий набор методов, позволяющих полностью интегрировать примитивные типы в иерархию объектов Java. Все они кратко рассматривается далее в главе.

Класс `Character`

Класс `Character` является оболочкой для значения `char`. Вот конструктор класса `Character`:

```
Character(char ch)
```

В аргументе `ch` указывается символ, который будет помещен в оболочку создаваемого объекта `Character`.

Однако, начиная с версии JDK 9, конструктор `Character` стал нерекомендуемым к употреблению, а начиная с JDK 16, он объявлен устаревшим и подлежащим удалению. В настоящее время для получения объекта `Character` настоятельно рекомендуется применять статический метод `valueOf()`:

```
static Character valueOf(char символ)
```

Метод `valueOf()` возвращает объект `Character`, содержащий внутри себя символ из `ch`.

Чтобы получить значение `char`, хранящееся в объекте `Character`, необходимо вызвать метод `charValue()`:

```
char charValue()
```

Он возвращает инкапсулированный символ.

Класс `Boolean`

Класс `Boolean` служит оболочкой для значений `boolean`. В нем определены следующие конструкторы:

```
Boolean(boolean boolValue)
Boolean(String boolString)
```

В первой версии конструктора аргумент `boolValue` должен быть либо `true`, либо `false`. Что касается второй версии конструктора, если аргумент `boolString` содержит строку `"true"` (в верхнем или нижнем регистре), тогда новый объект `Boolean` будет хранить значение `true`, а иначе — `false`.

Тем не менее, начиная с JDK 9, конструкторы `Boolean` были помечены как нерекомендуемые к использованию, а начиная с JDK 16, они стали устаревшими и подлежат удалению. На сегодняшний день для получения объекта `Boolean` настоятельно рекомендуется применять статический метод `valueOf()`, две версии которого показаны ниже:

```
static Boolean valueOf(boolean boolValue)
static Boolean valueOf(String boolString)
```

Каждая версия возвращает объект `Boolean`, служащий оболочкой для указанного значения.

Чтобы получить значение `boolean` из объекта `Boolean`, следует использовать метод `booleanValue()`:

```
boolean booleanValue()
```

Он возвращает эквивалент типа `boolean` для вызывающего объекта.

Оболочки числовых типов

Безусловно, наиболее часто применяемыми оболочками типов являются те, которые представляют числовые значения. Речь идет о `Byte`, `Short`, `Integer`, `Long`, `Float` и `Double`. Все оболочки числовых типов унаследованы от абстрактного класса `Number`. В классе `Number` объявлены методы, которые возвращают значение объекта в каждом из различных числовых форматов:

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

Например, метод `doubleValue()` возвращает значение объекта в виде `double`, `floatValue()` — в виде `float` и т.д. Указанные методы реализованы в каждой оболочке числового типа.

Во всех оболочках числовых типов определены конструкторы, которые позволяют создавать объект из заданного значения или строкового представления этого значения. Например, вот конструкторы, определенные в классе `Integer`:

```
Integer(int num)
Integer(String str)
```

Если в аргументе `str` не содержится допустимое числовое значение, тогда сгенерируется исключение `NumberFormatException`.

Однако, начиная с версии JDK 9, конструкторы оболочек числовых типов стали нерекомендуемыми к употреблению, а начиная с JDK 16, они объявлены устаревшими и подлежащими удалению. В настоящее время для получения объекта оболочки настоятельно рекомендуется использовать один из методов `valueOf()`. Метод `valueOf()` является статическим членом всех классов оболочек числовых типов и все числовые классы поддерживают формы,

которые преобразуют числовое значение или его строковое представление в объект. Например, вот две формы, поддерживаемые в Integer:

```
static Integer valueOf(int val)
static Integer valueOf(String valStr) throws NumberFormatException
```

В аргументе val указывается целочисленное значение, а в аргументе valStr — строка, которая представляет надлежащим образом сформатированное числовое значение в строковом виде. Каждая форма метода valueOf() возвращает объект Integer, содержащий внутри заданную величину. Ниже приведен пример:

```
Integer iOb = Integer.valueOf(100);
```

После выполнения этого оператора значение 100 будет представлено экземпляром Integer. Таким образом, объект iOb содержит в себе значение 100. В дополнение к только что показанным формам valueOf() целочисленные оболочки Byte, Short, Integer и Long также предоставляют форму, позволяющую указать систему счисления.

Все оболочки типов переопределяют метод toString(), который возвращает удобочитаемую форму значения, содержащегося внутри оболочки. Он позволяет выводить значение за счет передачи объекта оболочки типа, например, в println(), не требуя преобразования объекта в примитивный тип.

В показанной далее программе показано, как применять оболочку числового типа для инкапсуляции значения и последующего извлечения данного значения:

```
// Демонстрация использования оболочки числового типа.
class Wrap {
  public static void main(String[] args) {

    Integer iOb = Integer.valueOf(100);

    int i = iOb.intValue();

    System.out.println(i + " " + iOb);  // выводит 100 100
  }
}
```

В программе целочисленное значение 100 помещается внутрь объекта Integer по имени iOb, после чего путем вызова метода intValue() это значение получается и сохраняется в i.

Процесс инкапсуляции значения внутри объекта называется *упаковкой*. Таким образом, приведенная ниже строка в программе упаковывает значение 100 в объект Integer:

```
Integer iOb = Integer.valueOf(100);
```

Процесс извлечения значения из оболочки типа называется *распаковкой*. Например, значение внутри iOb распаковывается в программе посредством следующего оператора:

```
int i = iOb.intValue();
```

Та же самая общая процедура, которая использовалась в предыдущей программе для упаковки и распаковки значений, была доступна для применения, начиная с первоначальной версии Java. Тем не менее, на сегодняшний день в Java предлагается более рациональный подход, который описан далее в главе.

Автоупаковка

Современные версии Java включают два важных средства: *автоупаковку* и *автораспаковку*. Автоупаковка — это процесс, с помощью которого примитивный тип автоматически инкапсулируется (упаковывается) в эквивалентную ему оболочку типа всякий раз, когда требуется объект такого типа. Нет необходимости явно создавать объект. Автораспаковка — это процесс, при котором значение упакованного объекта автоматически извлекается (распаковывается) из оболочки типа, когда значение необходимо. Не придется вызывать методы вроде `intValue()` или `doubleValue()`.

Автоупаковка и автораспаковка значительно упрощают написание кода ряда алгоритмов, избавляя от утомительной ручной упаковки и распаковки значений, а также помогают предотвратить ошибки. Более того, они очень важны для обобщений, которые работают только с объектами. Наконец, автоупаковка существенно облегчает взаимодействие с инфраструктурой Collections Framework, описанной в части II.

Благодаря автоупаковке устраняется потребность в ручном создании объекта с целью помещения в него значения примитивного типа. Понадобится лишь присвоить это значение ссылке на оболочку типа, а компилятор Java самостоятельно создаст объект. Скажем, вот современный способ создания объекта `Integer` со значением 100:

```
Integer iOb = 100;       // автоупаковка значения int
```

Обратите внимание, что объект явно не упаковывается. Задачу автоматически решает компилятор Java.

Чтобы распаковать объект, нужно просто присвоить ссылку на него переменной примитивного типа. Например, для распаковки `iOb` можно использовать такую строку:

```
int i = iOb;             // автораспаковка
```

Обо всем остальном позаботится компилятор Java.

Ниже показана предыдущая программа, в которой теперь применяется автоупаковка/автораспаковка:

```
// Демонстрация работы автоупаковки/автораспаковки.
class AutoBox {
  public static void main(String[] args) {
    Integer iOb = 100;                    // автоупаковка значения int
    int i = iOb;                          // автораспаковка
    System.out.println(i + " " + iOb);    // выводит 100 100
  }
}
```

Автоупаковка и методы

В дополнение к простому случаю присваивания автоупаковка происходит всякий раз, когда примитивный тип должен быть преобразован в объект, а автораспаковка — когда объект должен быть преобразован в примитивный тип. Таким образом, автоупаковка/автораспаковка может быт инициирована при передаче аргумента методу или при возвращении методом значения. Например, взгляните на следующую программу:

```
// Автоупаковка/автораспаковка выполняется в отношении
// параметров и возвращаемого значения метода.
class AutoBox2 {
  // Принимает параметр типа Integer и возвращает значение int.
  static int m(Integer v) {
    return v;  // автораспаковка в int
  }

  public static void main(String[] args) {
    // Передать значение int в m() и присвоить возвращаемое значение
    // объекту Integer. Здесь аргумент 100 автоупаковывается
    // в объект Integer. Возвращаемое значение тоже автоупаковывается
    // в объект Integer.
    Integer iOb = m(100);

    System.out.println(iOb);
  }
}
```

Вот вывод, генерируемый программой:

```
100
```

Обратите внимание в программе, что `m()` принимает целочисленный параметр и возвращает результат типа `int`. Внутри `main()` методу `m()` передается значение `100`. Поскольку метод `m()` ожидает объект `Integer`, значение `100` автоматически упаковывается. Затем `m()` возвращает эквивалент типа `int` своего аргумента, приводя к автораспаковке `v`. Далее результирующее значение `int` присваивается `iOb` в `main()`, что приводит к автоупаковке возвращаемого значения `int`.

Автоупаковка/автораспаковка и выражения

Как правило, автоупаковка и автораспаковка происходят всякий раз, когда требуется преобразование в объект или из объекта, что относится и к выражениям. Внутри выражения числовой объект автоматически распаковывается. При необходимости результат выражения упаковывается заново. Например, рассмотрим показанную далее программу:

```
// Автоупаковка/автораспаковка происходит внутри выражений.
class AutoBox3 {
  public static void main(String[] args) {

    Integer iOb, iOb2;
    int i;
```

```
    iOb = 100;
    System.out.println("Исходное значение iOb: " + iOb);

    // В следующем операторе iOb автоматически распаковывается,
    // выполняется инкрементирование и результат заново
    // упаковывается в iOb.
    ++iOb;
    System.out.println("После ++iOb: " + iOb);

    // Здесь iOb распаковывается, выражение вычисляется,
    // результат заново упаковывается и присваивается iOb2.
    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 после вычисления выражения: " + iOb2);

    // Вычисляется то же самое выражение,
    // результат не упаковывается заново.
    i = iOb + (iOb / 3);
    System.out.println("i после вычисления выражения: " + i);
  }
}
```

Вот вывод:

```
Исходное значение iOb: 100
После ++iOb: 101
iOb2 после вычисления выражения: 134
i после вычисления выражения: 134
```

Обратите особое внимание в программе на следующую строку:

```
++iOb;
```

Данный оператор инкрементирует значение в `iOb`. Он работает так: объект `iOb` распаковывается, значение инкрементируется, а результат повторно упаковывается.

Автораспаковка также позволяет смешивать в выражении различные типы числовых объектов. После распаковки значений применяются стандартные повышения и преобразования. Скажем, приведенная ниже программа совершенно допустима:

```
class AutoBox4 {
  public static void main(String[] args) {

    Integer iOb = 100;
    Double dOb = 98.6;

    dOb = dOb + iOb;
    System.out.println("dOb после вычисления выражения: " + dOb);
  }
}
```

Вот вывод, генерируемый программой:

```
dOb после вычисления выражения: 198.6
```

Как видите, в сложении участвовали и объект `dOb` типа `Double`, и объект `iOb` типа `Integer`, а результат был заново упакован и сохранен в `dOb`.

Благодаря автораспаковке числовые объекты Integer можно использовать для управления оператором switch. В качестве примера взгляните на следующий фрагмент кода:

```
Integer iOb = 2;

switch(iOb) {
  case 1:
    System.out.println("один");
    break;
  case 2:
    System.out.println("два");
    break;
  default:
    System.out.println("ошибка");
}
```

При вычислении выражения в switch осуществляется распаковка объекта iOb и получение его значения int.

Как показывают примеры в программах, из-за автоупаковки/автораспаковки применение числовых объектов в выражении становится простым и интуитивно понятным. В ранних версиях Java такой код должен был включать приведения типов и вызовы методов, подобных intValue().

Автоупаковка/автораспаковка типов Boolean и Character

Как было описано ранее, в Java также предоставляются оболочки типов boolean и char — соответственно Boolean и Character, к которым также применяется автоупаковка/автораспаковка. Например, рассмотрим показанную далее программу:

```
// Автоупаковка/автораспаковка объектов Boolean и Character.
class AutoBox5 {
  public static void main(String[] args) {

    // Автоматически упаковать/распаковать значение boolean.
    Boolean b = true;

    // Ниже b автоматически распаковывается при использовании
    // в условном выражении, таком как if.
    if(b) System.out.println("b равно true");

    // Автоматически упаковать/распаковать значение char.
    Character ch = 'x';           // упаковать char
    char ch2 = ch;                // распаковать char

    System.out.println("ch2 равно " + ch2);
  }
}
```

Ниже приведен вывод:

```
b равно true
ch2 равно x
```

Самым важным аспектом, который следует отметить в программе, является автораспаковка b внутри условного выражения if. Вы наверняка помните о том, что вычисление условного выражения, управляющего оператором if, должно давать значение типа boolean. Значение boolean, содержащееся в b, автоматически распаковывается при вычислении условного выражения. Таким образом, при автоупаковке/автораспаковке объект Boolean можно использовать для управления оператором if.

Благодаря автораспаковке объект Boolean теперь также можно применять для управления любыми операторами цикла Java. Когда объект Boolean используется в качестве условного выражения в цикле while, for или do/while, он автоматически распаковывается в свой эквивалент boolean. Например, вот абсолютно корректный код:

```
Boolean b;
// ...
while(b) {  // ...
```

Автоупаковка/автораспаковка помогает предотвратить ошибки

Помимо удобства автоупаковка/автораспаковка также содействует в предотвращении ошибок. В качестве примера взгляните на следующую программу:

```
// Ошибка при ручной распаковке.
class UnboxingError {
  public static void main(String[] args) {

    Integer iOb = 1000;       // автоматически упаковать значение 1000

    int i = iOb.byteValue(); // вручную распаковать как byte!!!

    System.out.println(i);   // выводится не 1000!
  }
}
```

Вместо ожидаемого значения 1000 программа выводит -24! Дело в том, что значение внутри iOb распаковывается вручную путем вызова byteValue(), вызывая усечение значения, которое хранится в iOb и равно 1000. В итоге i присваивается "мусорное" значение -24. Автораспаковка предотвращает ошибки такого типа, поскольку значение в iOb будет всегда автоматически распаковываться в значение, совместимое с int.

В общем случае из-за того, что автоупаковка всегда создает надлежащий объект, а автораспаковка всегда производит корректное значение, процесс не может выдать неправильный тип объекта или значения. В тех редких ситуациях, когда нужен тип, отличающийся от созданного автоматическим процессом, все равно можно вручную упаковывать и распаковывать значения. Конечно, преимущества автоупаковки/автораспаковки утрачиваются. Таким образом, вы должны применять автоупаковку/автораспаковку. Именно так пишется современный код на Java.

Предостережение

По причине автоупаковки и автораспаковки у некоторых может возникнуть соблазн использовать исключительно такие объекты, как `Integer` или `Double`, полностью отказавшись от примитивных типов. Скажем, при автоупаковке/автораспаковке можно написать такой код:

```
// Неудачное применение автоупаковки/автораспаковки!
Double a, b, c;
a = 10.0;
b = 4.0;
c = Math.sqrt(a*a + b*b);
System.out.println("Гипотенуза равна " + c);
```

В приведенном примере объекты типа `Double` содержат значения, применяемые для вычисления гипотенузы прямоугольного треугольника. Хотя формально этот код корректен и действительно работает правильно, он демонстрирует крайне неудачное использование автоупаковки/автораспаковки. Он гораздо менее эффективен, чем эквивалентный код, написанный с применением примитивного типа `double`. Дело в том, что каждая автоупаковка и автораспаковка добавляют накладные расходы, которые отсутствуют в случае использования примитивного типа.

Вообще говоря, вам следует ограничить применение оболочек типов только теми случаями, когда объектное представление примитивного типа обязательно. Автоупаковка/автораспаковка не добавлялась в Java как "черный ход" для устранения примитивных типов.

Аннотации

Язык Java предлагает средство, позволяющее встраивать дополнительную информацию в файл исходного кода. Такая информация, называемая *аннотацией*, не меняет действия программы, оставляя семантику программы неизменной. Однако данная информация может использоваться разнообразными инструментами как во время разработки, так и во время развертывания. Например, аннотацию может обрабатывать генератор исходного кода. Для обозначения этого средства также применяется термин *метаданные*, но термин *аннотация* является наиболее описательным и часто используемым.

Основы аннотаций

Аннотация создается с помощью механизма, основанного на интерфейсе. Давайте начнем с примера. Вот объявление для аннотации по имени `MyAnno`:

```
// Простой тип аннотации.
@interface MyAnno {
  String str();
  int val();
}
```

Первым делом обратите внимание на символ @ перед ключевым словом interface. Он сообщает компилятору о том, что объявляется тип аннотации. Далее обратите внимание на два члена, str() и val(). Все аннотации состоят исключительно из объявлений методов. Тем не менее, вы не предоставляете тела для этих методов. Взамен их реализует компилятор Java. Более того, как вы увидите, методы во многом похожи на поля.

Аннотация не может содержать конструкцию extends. Однако все типы аннотаций автоматически расширяют интерфейс Annotation, который является суперинтерфейсом для всех аннотаций. Он объявлен в пакете java.lang.annotation и переопределяет методы hashCode(), equals() и toString(), определенные в классе Object. Интерфейс Annotation также задает annotationType(), который возвращает объект Class, представляющий вызывающую аннотацию.

Сразу после объявления аннотацию можно применять для снабжения чего-нибудь примечанием. Изначально аннотации можно было указывать только в объявлениях, с чего мы и начнем. (В версии JDK 8 добавлена возможность аннотирования сценариев использования типов, как будет показано далее в главе. Тем не менее, те же самые основные методы применимы к обоим видам аннотаций.) Любой тип объявления может иметь ассоциированную с ним аннотацию. Скажем, можно аннотировать классы, методы, поля, параметры и константы перечислений. Аннотировать допускается даже саму аннотацию. Во всех ситуациях аннотация предшествует остальной части объявления.

В случае применения аннотации ее элементам присваиваются значения. Например, вот пример применения аннотации MyAnno к объявлению метода:

```
// Аннотировать метод.
@MyAnno(str = "Пример аннотации", val = 100)
public static void myMeth() {  // ...
```

Здесь аннотация MyAnno связывается с методом myMeth(). Внимательно взгляните на синтаксис аннотации. За именем аннотации, которому предшествует символ @, находится заключенный в круглые скобки список инициализаций членов. Чтобы предоставить члену значение, его понадобится присвоить имени члена. Следовательно, в приведенном выше примере члену str аннотации MyAnno присваивается строка "Пример аннотации". Обратите внимание, что в этом присваивании после str никаких скобок нет. При предоставлении значения члену аннотации используется только его имя. Таким образом, в данном контексте члены аннотации выглядят как поля.

Указание политики хранения

Перед дальнейшим исследованием аннотаций необходимо обсудить *политики хранения аннотаций*. Политика хранения устанавливает момент, когда аннотация отбрасывается. В Java определены три такие политики, которые инкапсулированы внутри перечисления java.lang.annotation.RetentionPolicy — SOURCE, CLASS и RUNTIME.

- Аннотация с политикой хранения SOURCE удерживается только в файле исходного кода и на этапе компиляции отбрасывается.
- Аннотация с политикой хранения CLASS на этапе компиляции сохраняется в файле .class. Однако она не будет доступной через машину JVM во время выполнения.
- Аннотация с политикой хранения RUNTIME на этапе компиляции сохраняется в файле .class и доступна через машину JVM во время выполнения. Таким образом, политика RUNTIME обеспечивает наивысшее постоянство аннотаций.

На заметку! Аннотация на объявлении локальной переменной в файле .class не удерживается.

Политика хранения для аннотации указывается с применением одной из встроенных аннотаций Java: @Retention. Вот ее общая форма:

```
@Retention(retention-policy)
```

Здесь в аргументе retention-policy должна находиться одна из ранее обсуждавшихся констант перечисления. Если для аннотации не задана политика хранения, тогда используется стандартная политика CLASS.

В показанной далее версии MyAnno с применением @Retention указывается политика хранения RUNTIME. В результате аннотация MyAnno будет доступна машине JVM во время выполнения программы.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}
```

Получение аннотаций во время выполнения с использованием рефлексии

Аннотации предназначены главным образом для использования другими инструментами разработки или развертывания. Тем не менее, если для аннотаций определена политика хранения RUNTIME, тогда их можно запрашивать во время выполнения с помощью любой программы на Java через *рефлексию*. Рефлексия представляет собой средство, позволяющее получать информацию о классе во время выполнения. API-интерфейс рефлексии содержится в пакете java.lang.eflect. Существует несколько способов работы с рефлексией, но все они в главе рассматриваться не будут. Однако мы обсудим ряд примеров, применимых к аннотациям.

Первым шагом к использованию рефлексии является получение объекта Class, представляющего класс, аннотации которого вы хотите получить. Class — один из встроенных классов Java, определенный в пакете java.lang; он будет подробно описан в части II. Существуют различные способы получения объекта Class.

Один из самых простых способов предусматривает вызов метода getClass(), определенный в Object со следующей общей формой:

```
final Class<?> getClass()
```

Метод getClass() возвращает объект Class, который представляет вызывающий объект.

На заметку! Обратите внимание на символы <?> после Class в только что показанном объявлении метода getClass(). Они имеют отношение к средству обобщений в Java. Метод getClass() и несколько других методов, связанных с рефлексией, которые обсуждаются в этой главе, задействуют обобщения, описанные в главе 14. Тем не менее, для понимания фундаментальных принципов рефлексии понимать обобщения вовсе не обязательно.

После того, как объект Class получен, с помощью его методов можно просматривать информацию о различных элементах, объявленных в классе, в том числе об аннотациях. При желании извлечь аннотации, которые ассоциированы с конкретным элементом, объявленным в классе, сначала потребуется получить объект, представляющий данный элемент. Например, Class предлагает (помимо прочих) методы getMethod(), getField() и getConstructor(), предназначенные для получения информации о методе, поле и конструкторе соответственно. Эти методы возвращают объекты типа Method, Field и Constructor.

Чтобы понять сам процесс, давайте проработаем пример, в котором получаются аннотации, связанные с методом. Первым делом нужно получить объект Class, представляющий класс, а затем вызвать для этого объекта Class метод getMethod(), указав имя метода. Метод getMethod() имеет следующую общую форму:

```
Method getMethod(String methName, Class<?> ... paramTypes)
```

Имя метода передается в аргументе methName. Если у метода есть аргументы, то объекты Class, представляющие типы аргументов, также должны быть указаны в аргументе paramTypes. Обратите внимание, что paramTypes является параметром с переменным числом аргументов, т.е. можно задавать столько типов параметров, сколько необходимо, включая ноль. Метод getMethod() возвращает объект Method, представляющий метод. Если метод не может быть найден, тогда генерируется исключение NoSuchMethodException.

Вызывая метод getAnnotation(), из объекта Class, Method, Field или Constructor можно получить специфическую аннотацию, ассоциированную с этим объектом. Вот общая форма getAnnotation():

```
<A extends Annotation> getAnnotation(Class<A> annoType)
```

Здесь annoType — объект класса, представляющий интересующую аннотацию. Метод возвращает ссылку на аннотацию. С применением такой ссылки можно получить значения, связанные с членами аннотации. Метод возвращает null, если аннотация не найдена, что происходит, когда аннотация не имеет политики хранения RUNTIME.

Ниже приведена программа, в которой все показанные ранее части собраны вместе и которая использует рефлексию для отображения аннотации, ассоциированной с методом:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Объявление аннотации типа.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

class Meta {

  // Аннотировать метод.
  @MyAnno(str = "Пример аннотации", val = 100)
  public static void myMeth() {
    Meta ob = new Meta();

    // Получить аннотацию для этого метода
    // и отобразить значения ее членов.

    try {
      // Для начала получить объект Class,
      // который представляет данный класс.
      Class<?> c = ob.getClass();

      // Теперь получить объект Method,
      // который представляет данный метод.
      Method m = c.getMethod("myMeth");

      // Далее получить аннотацию для этого класса.
      MyAnno anno = m.getAnnotation(MyAnno.class);

      // В заключение вывести значения.
      System.out.println(anno.str() + " " + anno.val());
    } catch (NoSuchMethodException exc) {
      System.out.println("Метод не найден.");
    }
  }

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

Вот вывод, генерируемый программой:

```
Пример аннотации 100
```

В программе применяется рефлексия, как было описано ранее, для получения и отображения значений `str` и `val` в аннотации `MyAnno`, ассоциированной с методом `myMeth()` в классе `Meta`. Стоит обратить особое внимание на два аспекта. Первый из них — выражение `MyAnno.class` в следующей строке:

```
MyAnno anno = m.getAnnotation(MyAnno.class);
```

Результатом вычисления данного выражения является объект `Class` типа `MyAnno`, т.е. аннотация. Такая конструкция называется *литералом класса*. Этот тип выражения можно использовать всякий раз, когда необходим объект `Class` известного класса. Например, с помощью показанного далее оператора можно было бы получить объект `Class` для `Meta`:

```
Class<?> c = Meta.class;
```

Конечно, такой подход работает, только если имя класса объекта известно заранее, что далеко не всегда так. В общем случае литерал класса можно получать для классов, интерфейсов, примитивных типов и массивов. (Не забывайте, что синтаксис <?> относится к средству обобщений Java, которое обсуждается в главе 14.)

Вторым интересным аспектом является способ получения значений `str` и `val`, когда они выводятся следующей строкой:

```
System.out.println(anno.str() + " " + anno.val());
```

Обратите внимание, что они вызываются с применением синтаксиса вызова методов. Тот же самый подход используется всякий раз, когда требуется значение члена аннотации.

Второй пример использования рефлексии

В предыдущем примере метод `myMeth()` не принимает параметры. Таким образом, при вызове `getMethod()` передается только имя `myMeth`. Однако чтобы получить метод с параметрами, в качестве аргументов `getMethod()` понадобится указать объекты класса, представляющие типы этих параметров. Например, вот слегка измененная версия предыдущей программы:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

class Meta {

  // Метод myMeth теперь принимает два аргумента.
  @MyAnno(str = "Два параметра", val = 19)
  public static void myMeth(String str, int i)
  {
    Meta ob = new Meta();

    try {
      Class<?> c = ob.getClass();

      // Здесь указываются типы параметров.
      Method m = c.getMethod("myMeth", String.class, int.class);

      MyAnno anno = m.getAnnotation(MyAnno.class);

      System.out.println(anno.str() + " " + anno.val());
    } catch (NoSuchMethodException exc) {
```

```
      System.out.println("Метод не найден.");
    }
  }

  public static void main(String[] args) {
    myMeth("тест", 10);
  }
}
```

Ниже показан вывод, генерируемый программой:

```
Два параметра 19
```

В данной версии программы метод `myMeth()` принимает параметры типа `String` и `int`. Для получения информации об этом методе необходимо вызвать метод `getMethod()` следующим образом:

```
Method m = c.getMethod("myMeth", String.class, int.class);
```

Объекты `Class`, представляющие типы `String` и `int`, передаются в виде дополнительных аргументов.

Получение всех аннотаций

Чтобы получить все аннотации с политикой хранения `RUNTIME`, ассоциированные с элементом, можно вызвать на данном элементе метод `getAnnotations()`, который имеет приведенную ниже общую форму:

```
Annotation[] getAnnotations()
```

Метод `getAnnotations()` возвращает массив аннотаций и может быть вызван на объектах типа `Class`, `Method`, `Constructor` и `Field` (помимо прочих).

```
// Отображение всех аннотаций для класса и метода.
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

@Retention(RetentionPolicy.RUNTIME)
@interface What {
  String description();
}

@What(description = "Аннотация класса")
@MyAnno(str = "Meta2", val = 99)
class Meta2 {

  @What(description = "Аннотация метода")
  @MyAnno(str = "Testing", val = 100)
  public static void myMeth() {
    Meta2 ob = new Meta2();

    try {
      Annotation[] annos = ob.getClass().getAnnotations();
```

```
      // Отобразить все аннотации для Meta2.
      System.out.println("Все аннотации для класса Meta2:");
      for(Annotation a : annos)
        System.out.println(a);

      System.out.println();

      // Отобразить все аннотации для myMeth.
      Method m = ob.getClass().getMethod("myMeth");
      annos = m.getAnnotations();
      System.out.println("Все аннотации для метода myMeth:");
      for(Annotation a : annos)
        System.out.println(a);
    } catch (NoSuchMethodException exc) {
      System.out.println("Метод не найден.");
    }
  }
  public static void main(String[] args) {
    myMeth();
  }
}
```

Вот вывод:

```
Все аннотации для класса Meta2:
@What(description=Аннотация класса)
@MyAnno(str=Meta2, val=99)

Все аннотации для метода myMeth:
@What(description=Аннотация метода)
@MyAnno(str=Testing, val=100)
```

Метод `getAnnotations()` используется в программе для получения массива всех аннотаций, ассоциированных с классом `Meta2` и методом `myMeth()`. Как объяснялось ранее, `getAnnotations()` возвращает массив объектов `Annotation`. Вспомните, что тип `Annotation` является суперинтерфейсом для всех интерфейсов аннотаций и переопределяет метод `toString()` в `Object`. Таким образом, при выводе ссылки на объект `Annotation` вызывается его метод `toString()` для создания строки, описывающей аннотацию, что было видно в предыдущем выводе.

Интерфейс AnnotatedElement

Методы `getAnnotation()` и `getAnnotations()`, применяемые в предшествующих примерах, определены в интерфейсе `AnnotatedElement` из пакета `java.lang.reflect`. Он поддерживает рефлексию для аннотаций и реализован среди прочего классами `Method`, `Field`, `Constructor`, `Class` и `Package`.

Помимо `getAnnotation()` и `getAnnotations()` в `AnnotatedElement` определено несколько других методов.

Два метода доступны с тех пор, как аннотации были первоначально добавлены в Java. Первый — `getDeclaredAnnotations()`, который имеет следующую общую форму:

```
Annotation[] getDeclaredAnnotations()
```

Метод getDeclaredAnnotations() возвращает все неунаследованные аннотации, присутствующие в вызывающем объекте. Второй метод — isAnnotationPresent() с такой общей формой:

```
default boolean isAnnotationPresent(Class<? extends Annotation> annoType)
```

Метод isAnnotationPresent() возвращает true, если аннотация, указанная в аргументе annoType, ассоциирована с вызывающим объектом. В противном случае он возвращает false. В версии JDK 8 к методам getDeclaredAnnotations() и isAnnotationPresent() были добавлены методы getDeclaredAnnotation(), getAnnotationsByType() и getDeclaredAnnotationsByType(). Последние два автоматически работают с повторяющейся аннотацией (повторяющиеся аннотации обсуждаются в конце главы).

Использование стандартных значений

Для членов аннотации можно задавать стандартные значения, которые будут использоваться, если при применении аннотации не указано значение. Стандартное значение устанавливается путем добавления к объявлению члена member() конструкции default, имеющей следующую общую форму:

```
type member() default value;
```

Здесь значение value должно иметь тип, совместимый с типом, который указан в type.

Ниже показана аннотация @MyAnno, переписанная с целью включения стандартных значений:

```
// Объявление типа аннотации, включающее стандартные значения.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str() default "Тест";
  int val() default 9000;
}
```

В объявлении str получает стандартное значение "Тест", а val — 9000. Таким образом, при использовании @MyAnno не нужно указывать ни одно из значений. Тем не менее, в случае необходимости одному или обоим могут быть присвоены значения.

Ниже приведены четыре способа применения @MyAnno:

```
@MyAnno()                  // используются стандартные значения для str и val
@MyAnno(str = "строка")    // используется стандартное значение для val
@MyAnno(val = 100)         // используется стандартное значение для str
@MyAnno(str = "Тест", val = 100)  // стандартные значение не используются
```

В следующей программе демонстрируется использование стандартных значений в аннотации:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Объявление типа аннотации, включающее стандартные значения.
```

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str() default "Тест";
  int val() default 9000;
}

class Meta3 {

  // Аннотировать метод с использованием стандартных значений.
  @MyAnno()
    public static void myMeth() {
      Meta3 ob = new Meta3();

      // Получить аннотацию для этого метода
      // и вывести значения ее членов.
      try {
        Class<?> c = ob.getClass();
        Method m = c.getMethod("myMeth");

        MyAnno anno = m.getAnnotation(MyAnno.class);

        System.out.println(anno.str() + " " + anno.val());
      } catch (NoSuchMethodException exc) {
        System.out.println("Метод не найден.");
      }
    }

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

Вот вывод:

```
Тест 9000
```

Маркерные аннотации

Маркерная аннотация является специальным видом аннотации, не содержащим членов. Ее единственная цель — пометить элемент. Таким образом, вполне достаточно наличия данной аннотации. Лучший способ определить наличие маркерной аннотации предусматривает применение метода `isAnnotationPresent()`, который определен в интерфейсе `AnnotatedElement`.

Далее приведен пример, в котором используется маркерная аннотация. Поскольку такая аннотация не содержит членов, достаточно просто выяснить, присутствует она или же отсутствует.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Маркерная аннотация.
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }

class Marker {

  // Аннотировать метод, используя маркерную аннотацию.
  // Обратите внимание, что скобки () не требуются.
```

```
  @MyMarker
  public static void myMeth() {
    Marker ob = new Marker();

    try {
      Method m = ob.getClass().getMethod("myMeth");

      // Выяснить, присутствует ли аннотация.
      if(m.isAnnotationPresent(MyMarker.class))
        System.out.println("MyMarker присутствует.");
    } catch (NoSuchMethodException exc) {
      System.out.println("Метод не найден.");
    }
  }

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

Вывод, генерируемый программой, подтверждает присутствие `@MyMarker`:

```
MyMarker присутствует.
```

Обратите в программе внимание, что указывать круглые скобки после аннотации `@MyMarker` в случае ее использования не требуется, т.е. `@MyMarker` применяется просто по своему имени:

```
@MyMarker
```

К маркерной аннотации можно добавить пустой набор круглых скобок, что не считается ошибкой, но они не обязательны.

Одноэлементные аннотации

Одноэлементная аннотация содержит только один член. Она работает как нормальная аннотация за исключением того, что позволяет использовать сокращенную форму для указания значения члена. Когда присутствует только один член, при применении аннотации можно просто указать значение для него — указывать имя члена не нужно. Однако для использования такого сокращения именем члена должно быть `value`.

Ниже показан пример создания и применения одноэлементной аннотации:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Одноэлементная аннотация.
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
  int value();  // именем члена должно быть value
}

class Single {

  // Аннотировать метод, используя одноэлементную аннотацию.
  @MySingle(100)
  public static void myMeth() {
    Single ob = new Single();
```

```
    try {
      Method m = ob.getClass().getMethod("myMeth");
      MySingle anno = m.getAnnotation(MySingle.class);
      System.out.println(anno.value()); // выводит 100
    } catch (NoSuchMethodException exc) {
      System.out.println("Метод не найден.");
    }
  }
  public static void main(String[] args) {
    myMeth();
  }
}
```

Программа вполне ожидаемо отображает значение 100, а @MySingle используется в ней для аннотирования метода myMeth():

```
@MySingle(100)
```

Обратите внимание, что указывать value = не обязательно.

Синтаксис с одним значением можно использовать при применении аннотации, которая имеет другие члены, но все эти другие члены должны быть снабжены стандартными значениями. Например, в следующем фрагменте кода добавлен член xyz со стандартным значением, равным нулю:

```
@interface SomeAnno {
  int value();
  int xyz() default 0;
}
```

В ситуациях, когда для xyz желательно использовать стандартное значение, можно применить @SomeAnno, просто указав значение value с помощью синтаксиса с одним членом:

```
@SomeAnno(88)
```

В этом случае стандартным значением xyz будет ноль, а value получит значение 88. Конечно, чтобы задать другое значение для xyz, необходимо явно указать оба члена:

```
@SomeAnno(value = 88, xyz = 99)
```

Не забывайте, что всякий раз, когда используется одноэлементная аннотация, именем элемента должно быть value.

Встроенные аннотации

В Java определено множество встроенных аннотаций. Большинство из них специализированы, но девять являются аннотациями общего назначения. Четыре аннотации импортируются из пакета java.lang.annotation: @Retention, @Documented, @Target и @Inherited.

Пять аннотаций — @Override, @Deprecated, @FunctionalInterface, @SafeVarargs и @SuppressWarnings — входят в состав пакета java.lang. Все они описаны ниже.

На заметку! Пакет `java.lang.annotation` также включает аннотации `Repeatable` и `Native`; из них `Repeatable` поддерживает повторяющиеся аннотации, рассматриваемые позже в главе, а `Native` позволяет аннотировать поле, доступ к которому может получать машинный код.

@Retention

Аннотация `@Retention` рассчитана только на применение к другой аннотации. Она определяет политику хранения, как было описано ранее в главе.

@Documented

Аннотация `@Documented` представляет собой маркерный интерфейс, который сообщает инструменту о том, что аннотация должна быть документирована. Она предназначена для использования только в качестве аннотации к объявлению аннотации.

@Target

С помощью аннотации `@Target` задаются типы элементов, к которым может применяться аннотация. Она спроектирована для использования только в качестве аннотации к другой аннотации. Аннотация `@Target` принимает один аргумент, представляющий собой массив констант перечисления `ElementType`. В этом аргументе указываются типы объявлений, к которым может применяться аннотация. Константы описаны в табл. 12.1 вместе с типами объявлений, которым они соответствуют.

Таблица 12.1. Константы перечисления `ElementType`

Целевая константа	Объявление, к которому может быть применена аннотация
`ANNOTATION_TYPE`	Другая аннотация
`CONSTRUCTOR`	Конструктор
`FIELD`	Поле
`LOCAL_VARIABLE`	Локальная переменная
`METHOD`	Метод
`MODULE`	Модуль
`PACKAGE`	Пакет
`PARAMETER`	Параметр
`RECORD_COMPONENT`	Компонент записи (добавлен в JDK 16)
`TYPE`	Класс, интерфейс или перечисление
`TYPE_PARAMETER`	Параметр типа
`TYPE_USE`	Использование типа

В аннотации `@Target` можно указывать одно или несколько значений из табл. 12.1. Несколько значений задаются в виде списка, заключенного в фигурные скобки. Например, вот как можно указать о том, что аннотация `@Target` применяется только к полям и локальным переменным:

```
@Target( { ElementType.FIELD, ElementType.LOCAL_VARIABLE } )
```

Если `@Target` отсутствует, тогда аннотацию можно использовать в любом объявлении. По этой причине часто рекомендуется явно указывать цель или цели, чтобы четко обозначить предполагаемое применение аннотации.

@Inherited

`@Inherited` является маркерной аннотацией, которую можно использовать только в объявлении другой аннотации. Кроме того, `@Inherited` влияет только на аннотации, которые будут применяться в объявлениях классов. Указание `@Inherited` приводит к тому, что аннотация для суперкласса наследуется подклассом. Следовательно, при запросе у подкласса конкретной аннотации, если эта аннотация отсутствует в подклассе, то проверяется его суперкласс. Если же аннотация присутствует в суперклассе и снабжена аннотацией `@Inherited`, тогда она будет возвращена.

@Override

`@Override` — маркерная аннотация, которую можно использовать только для методов. Метод с аннотацией `@Override` должен переопределять метод из суперкласса, иначе возникнет ошибка на этапе компиляции. Она применяется для гарантирования того, что метод суперкласса действительно переопределен, а не просто перегружен.

@Deprecated

Аннотация `@Deprecated` указывает, что объявление устарело и не рекомендуется к употреблению. Начиная с JDK 9, аннотация `@Deprecated` также позволяет указать версию Java, в которой было заявлено об устаревании, и планируется ли удаление устаревшего элемента.

@FunctionalInterface

`@FunctionalInterface` представляет собой маркерную аннотацию, предназначенную для использования в интерфейсах. Она указывает на то, что аннотированный интерфейс является функциональным интерфейсом. *Функциональный интерфейс* — это интерфейс, который содержит один и только один абстрактный метод. Функциональные интерфейсы задействованы в лямбда-выражениях. (Подробные сведения о функциональных интерфейсах и лямбда-выражениях ищите в главе 15.) Если аннотированный интерфейс не относится к функциональным, тогда будет сообщено об ошибке на этапе компиляции. Важно понимать, что аннотация `@FunctionalInterface` не нужна для создания функционального интерфейса. Любой интерфейс, имеющий в

точности один абстрактный метод, по определению будет функциональным интерфейсом. Таким образом, аннотация `@FunctionalInterface` носит чисто информационный характер.

@SafeVarargs

`@SafeVarargs` — маркерная аннотация, которую можно применять к методам и конструкторам. Она указывает на отсутствие небезопасных действий, связанных с аргументом переменной длины. Аннотация `@SafeVarargs` используется для подавления непроверяемых предупреждений в безопасном в остальном коде, поскольку он относится к нематериализуемым (non-reifiable) типам аргументов переменной длины и созданию экземпляров параметризованного массива. (Нематериализуемый тип, по сути, представляет собой обобщенный тип. Обобщенные типы обсуждаются в главе 14.) Она должна применяться только к методам или конструкторам с аргументами переменной длины. Методы также обязаны быть `static`, `final` или `private`.

@SuppressWarnings

Аннотация `@SuppressWarnings` указывает, что одно или несколько предупреждений, которые могут быть выданы компилятором, должны быть подавлены. Предупреждения, подлежащие подавлению, задаются по имени в строковой форме.

Аннотации типов

Как упоминалось ранее, первоначально аннотации были разрешены только в объявлениях. Тем не менее, современные версии Java позволяют указывать аннотации в большинстве случаев использования типов. Такое расширенное свойство аннотаций называется *аннотациями типов*. Скажем, можно аннотировать возвращаемый тип метода, тип `this` внутри метода, приведение, уровни массива, унаследованный класс и конструкцию `throws`. Вдобавок можно аннотировать обобщенные типы, включая границы параметров обобщенного типа и аргументы обобщенного типа. (Обобщения рассматриваются в главе 14.)

Аннотации типов важны, поскольку они позволяют инструментам выполнять дополнительные проверки кода, чтобы предотвратить ошибки. Важно понимать, что компилятор `javac`, как правило, не будет проводить такие проверки самостоятельно. Для этой цели используется отдельный инструмент, хотя такой инструмент может функционировать как подключаемый модуль компилятора.

Аннотация типа должна включать `ElementType.TYPE_USE` в качестве цели. (Вспомните, что согласно приведенным ранее объяснениям, допустимые цели аннотаций указываются с помощью аннотации `@Target`.) Аннотация типа применяется к типу, которому она предшествует. Например, если предположить, что аннотация типа имеет имя `@TypeAnno`, то разрешено записывать так:

```
void myMeth() throws @TypeAnno NullPointerException {  // ...
```

Здесь @TypeAnno аннотирует NullPointerException в конструкции throws.

Можно также аннотировать тип this (называемый *получателем*). Как вам известно, это неявный аргумент для всех методов экземпляра, и он относится к вызываемому объекту. Аннотирование его типа требует использования другого средства, которое изначально не было частью Java. Начиная с JDK 8, this можно явно объявлять в качестве первого параметра метода. В таком объявлении типом this должен быть класс, например:

```
class SomeClass {
  int myMeth(SomeClass this, int i, int j) {  // ...
```

Поскольку myMeth() — метод, определенный в SomeClass, его типом является SomeClass. Теперь с применением этого объявления можно аннотировать тип this. Скажем, снова предполагая наличие аннотации типа @TypeAnno, следующий код будет допустимым:

```
int myMeth(@TypeAnno SomeClass this, int i, int j) {  // ...
```

Важно понимать, что объявлять this нет никакой необходимости, если вы не аннотируете this. (Если аргумент this не объявляется, то он все равно передается неявно, как было всегда.) Кроме того, явное объявление this не меняет никаких аспектов сигнатуры метода, т.к. по умолчанию this объявляется неявно. Стоит повториться: вы будете объявлять this, только если хотите применить к нему аннотацию типа. В случае объявления this он *обязан* быть первым аргументом.

В следующей программе демонстрируется несколько мест, где можно использовать аннотацию типа. В ней определен ряд аннотаций, часть которых предназначена для аннотирования типов. Имена и цели аннотаций приведены в табл. 12.2.

Таблица 12.2. Имена и цели аннотаций типов

Аннотация	Цель
@TypeAnno	ElementType.TYPE_USE
@MaxLen	ElementType.TYPE_USE
@NotZeroLen	ElementType.TYPE_USE
@Unique	ElementType.TYPE_USE
@What	ElementType.TYPE_PARAMETER
@EmptyOK	ElementType.FIELD
@Recommended	ElementType.METHOD

Обратите внимание, что @EmptyOK, @Recommended и @What не являются аннотациями типов. Они включены в целях сравнения. Особый интерес представляет аннотация @What, которая применяется для аннотирования объявления параметра обобщенного типа. Все случаи использования описаны в комментариях внутри программы.

```
// Демонстрация использования нескольких аннотаций типов.
import java.lang.annotation.*;
import java.lang.reflect.*;

// Маркерная аннотация может применяться к типу.
@Target(ElementType.TYPE_USE)
@interface TypeAnno { }

// Вторая маркерная аннотация, которая может быть применена к типу.
@Target(ElementType.TYPE_USE)
@interface NotZeroLen {
}
// Третья маркерная аннотация, которая может быть применена к типу.
@Target(ElementType.TYPE_USE)
@interface Unique { }

// Параметризованная аннотация, которая может быть применена к типу.
@Target(ElementType.TYPE_USE)
@interface MaxLen {
  int value();
}
// Аннотация, которая может быть применена к параметру типа.
@Target(ElementType.TYPE_PARAMETER)
@interface What {
  String description();
}
// Аннотация, которая может быть применена к объявлению поля.
@Target(ElementType.FIELD)
@interface EmptyOK { }

// Аннотация, которая может быть применена к объявлению метода.
@Target(ElementType.METHOD)
@interface Recommended { }

// Использовать аннотацию на параметре типа.
class TypeAnnoDemo<@What(description = "Обобщенный тип данных") T> {

  // Использовать аннотацию типа для конструктора.
  public @Unique TypeAnnoDemo() {}

  // Аннотировать тип (в этом случае String), не поле.
  @TypeAnno String str;

  // Аннотировать поле test.
  @EmptyOK String test;

  // Использовать аннотацию типа для аннотирования this (получателя).
  public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {
    return 10;
  }

  // Аннотировать возвращаемый тип.
  public @TypeAnno Integer f2(int j, int k) {
    return j+k;
  }

  // Аннотировать объявление метода.
  public @Recommended Integer f3(String str) {
    return str.length() / 2;
  }
```

```
  // Использовать аннотацию типа с конструкцией throws.
  public void f4() throws @TypeAnno NullPointerException {
    // ...
  }

  // Аннотировать уровни массива.
  String @MaxLen(10) [] @NotZeroLen [] w;

  // Аннотировать тип элементов массива.
  @TypeAnno Integer[] vec;

  public static void myMeth(int i) {

    // Использовать аннотацию типа для аргумента типа.
    TypeAnnoDemo<@TypeAnno Integer> ob =
      new TypeAnnoDemo<@TypeAnno Integer>();

    // Использовать аннотацию типа для операции new.
    @Unique TypeAnnoDemo<Integer> ob2 =
      new @Unique TypeAnnoDemo<Integer>();

    Object x = Integer.valueOf(10);
    Integer y;

    // Использовать аннотацию типа для приведения.
    y = (@TypeAnno Integer) x;
  }

  public static void main(String[] args) {
    myMeth(10);
  }

  // Использовать аннотацию типа для конструкции наследования.
  class SomeClass extends @TypeAnno TypeAnnoDemo<Boolean> {}
}
```

Хотя то, к чему относится большинство аннотаций в предыдущей программе, совершенно ясно, четыре варианта использования требуют некоторого обсуждения. Прежде всего, сравним аннотацию возвращаемого типа метода с аннотацией объявления метода. Обратите особое внимание в программе на следующие два объявления методов:

```
// Аннотировать возвращаемый тип.
public @TypeAnno Integer f2(int j, int k) {
  return j+k;
}

// Аннотировать объявление метода.
public @Recommended Integer f3(String str) {
  return str.length() / 2;
}
```

В обоих случаях аннотация предшествует возвращаемому типу метода (`Integer`). Однако эти две аннотации аннотируют два разных элемента. В первом случае аннотация `@TypeAnno` аннотирует возвращаемый тип метода `f2()`. Причина в том, что в качестве цели аннотации `@TypeAnno` указано `ElementType.TYPE_USE` и потому ее можно применять для аннотирования использования типов. Во втором случае аннотация `@Recommended` аннотирует само объявление метода, т.к. целью `@Recommended` является `ElementType.METHOD`.

В результате @Recommended применяется к объявлению, а не к возвращаемому типу. Следовательно, спецификация цели позволяет устранить то, что на первый взгляд кажется неоднозначностью между аннотациями объявления и возвращаемого типа метода.

Еще один момент, касающийся аннотирования возвращаемого типа метода: аннотировать возвращаемый тип void нельзя.

Второй интересный момент связан с аннотациями полей:

```
// Аннотировать тип (в этом случае String), не поле.
@TypeAnno String str;

// Аннотировать поле test.
@EmptyOK String test;
```

Аннотация @TypeAnno аннотирует тип String, а @EmptyOK — поле test. Несмотря на то что обе аннотации предшествуют всему объявлению, их цели различаются в зависимости от типа целевого элемента. Если аннотация имеет цель ElementType.TYPE_USE, тогда аннотируется тип. Если в качестве цели указано ElementType.FIELD, то аннотируется поле. Таким образом, ситуация аналогична только что описанной для методов, и никакой неоднозначности нет. Тот же самый механизм устраняет неоднозначность аннотаций на локальных переменных.

Теперь обратите внимание на аннотирование this (получателя):

```
public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {
```

Здесь this указывается в качестве первого параметра и имеет тип TypeAnnoDemo (класс, членом которого является метод f()). Как объяснялось ранее, в объявлении метода экземпляра можно явно указывать параметр this, чтобы применить к нему аннотацию типа.

Наконец, взгляните на аннотирование уровней массива:

```
String @MaxLen(10) [] @NotZeroLen [] w;
```

В этом объявлении @MaxLen аннотирует тип первого уровня, а @NotZeroLen — тип второго уровня. В следующем объявлении аннотируется тип элементов Integer:

```
@TypeAnno Integer[] vec;
```

Повторяющиеся аннотации

Начиная с JDK 8, аннотацию разрешено многократно применять к одному и тому же элементу. Такие аннотации называются *повторяющимися*. Чтобы аннотацию можно было повторять, ее потребуется снабдить аннотацией @Repeatable, определенной в пакете java.lang.annotation, и указать в ее поле value тип *контейнера* для повторяющейся аннотации. Контейнер задается в виде аннотации, для которой поле value представляет собой массив типа повторяющейся аннотации. Таким образом, для создания повторяющейся аннотации необходимо создать контейнерную аннотацию, после чего указать этот тип аннотации в качестве аргумента аннотации @Repeatable.

Чтобы получить доступ к повторяющимся аннотациям с помощью метода вроде getAnnotation(), будет использоваться контейнерная аннотация, а не сама повторяющаяся аннотация. Такой подход демонстрируется в следующей программе, где показанная ранее версия MyAnno преобразуется в повторяющуюся аннотацию, которая затем применяется.

```
// Демонстрация использования повторяющихся аннотаций.
import java.lang.annotation.*;
import java.lang.reflect.*;

// Сделать аннотацию MyAnno повторяющейся.
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(MyRepeatedAnnos.class)
@interface MyAnno {
  String str() default "Тест";
  int val() default 9000;
}

// Это контейнерная аннотация.
@Retention(RetentionPolicy.RUNTIME)
@interface MyRepeatedAnnos {
  MyAnno[] value();
}

class RepeatAnno {

  // Повторить аннотацию MyAnno для метода myMeth().
  @MyAnno(str = "Первая аннотация", val = -1)
  @MyAnno(str = "Вторая аннотация", val = 100)
  public static void myMeth(String str, int i)
  {
    RepeatAnno ob = new RepeatAnno();

    try {
      Class<?> c = ob.getClass();

      // Получить аннотации для метода myMeth().
      Method m = c.getMethod("myMeth", String.class, int.class);

      // Отобразить повторяющиеся аннотации MyAnno.
      Annotation anno = m.getAnnotation(MyRepeatedAnnos.class);
      System.out.println(anno);
    } catch (NoSuchMethodException exc) {
      System.out.println("Метод не найден.");
    }
  }

  public static void main(String[] args) {
    myMeth("тест", 10);
  }
}
```

Ниже показан вывод, генерируемый программой:

```
@MyRepeatedAnnos(value={@MyAnno(val=-1, str="Первая аннотация"),
@MyAnno(val=100, str="Вторая аннотация")})
```

Как объяснялось ранее, чтобы аннотацию MyAnno можно было повторять, она должна быть снабжена аннотацией @Repeatable, которая задает ее кон-

тейнерную аннотацию. Контейнерная аннотация имеет имя MyRepeatedAnnos. Программа получает доступ к повторяющимся аннотациям, вызывая метод getAnnotation() с передачей ему класса контейнерной аннотации, а не самой повторяющейся аннотации. В выводе видно, что повторяющиеся аннотации отделяются друг от друга запятыми. По отдельности они не возвращаются.

Другой способ получения повторяющихся аннотаций предусматривает использование одного из методов в AnnotatedElement, который способен работать напрямую с повторяющейся аннотацией — getAnnotationsByType() и getDeclaredAnnotationsByType(). Вот общая форма первого метода:

```
default <T extends Annotation> T[] getAnnotationsByType(Class<T> annoType)
```

Метод getAnnotationsByType() возвращает массив аннотаций типа annoType, ассоциированных с вызывающим объектом. Если аннотации отсутствуют, тогда массив будет иметь нулевую длину.

Рассмотрим пример. С учетом предыдущей программы в следующей кодовой последовательности метод getAnnotationsByType() используется для получения повторяющихся аннотаций MyAnno:

```
Annotation[] annos = m.getAnnotationsByType(MyAnno.class);
for(Annotation a : annos)
  System.out.println(a);
```

Методу getAnnotationsByType() передается тип повторяющейся аннотации MyAnno. Возвращаемый массив содержит все экземпляры MyAnno, связанные с myMeth(), которых в этом примере два. К каждой повторяющейся аннотации можно получать доступ через ее индекс в массиве. В данном случае каждая аннотация MyAnno отображается через цикл for в стиле "for-each".

Некоторые ограничения

Существует несколько ограничений, применяемых к объявлениям аннотаций. Во-первых, ни одна аннотация не может быть унаследована от другой. Во-вторых, все методы, объявленные аннотацией, не должны принимать параметры. Кроме того, они обязаны возвращать один из перечисленных далее типов:

- примитивный тип, такой как int или double;
- объект типа String или Class;
- объект типа enum;
- объект типа другой аннотации;
- массив одного из допустимых типов.

Аннотации не могут быть обобщенными. Другими словами, они не могут принимать параметры типа. (Обобщения описаны в главе 14.) Наконец, в методах аннотаций нельзя указывать конструкцию throws.

ГЛАВА

13 Ввод-вывод, оператор try с ресурсами и другие темы

В настоящей главе представлен один из самых важных пакетов Java — `java.io`, который поддерживает базовую систему ввода-вывода Java, включая файловый ввод-вывод. По ддержка ввода-вывода обеспечивается основными библиотеками Java API, а не ключевыми словами языка. По указанной причине подробное обсуждение данной темы можно найти в части II книги, где рассматриваются несколько пакетов Java API. Здесь представлена основа этой важной подсистемы, чтобы вы могли увидеть, как она вписывается в более широкий контекст программирования на Java и исполняющей среды Java. В главе также исследуются оператор `try` с ресурсами и несколько дополнительных ключевых слов Java: `transient`, `volatile`, `instanceof`, `native`, `strictfp` и `assert`. В заключение обсуждается статическое импортирование и описано еще одно использование ключевого слова `this`.

Основы ввода-вывода

Вероятно, вы заметили при чтении предшествующих 12 глав, что в примерах программ ввод-вывод практически не применялся. Фактически ни один из методов ввода-вывода кроме `print()` и `println()` не использовался сколь-нибудь значительно. Причина проста: большинство реальных приложений Java не являются текстовыми консольными программами. Напротив, они представляют собой либо графически-ориентированные программы, которые для взаимодействия с пользователем применяют одну из инфраструктур Java для построения графических пользовательских интерфейсов, такую как Swing, либо веб-приложения. Хотя текстовые консольные программы превосходны в качестве обучающих примеров, как правило, с ними не связаны какие-то важные сценарии использования Java в реальном мире. Кроме того, поддержка консольного ввода-вывода в Java ограничена и несколько неудобна в применении — даже в простых примерах программ. Текстовый консольный ввод-вывод просто не настолько полезен в реальном программировании на Java.

Несмотря на то, что упоминалось в предыдущем абзаце, язык Java обеспечивает мощную и гибкую поддержку ввода-вывода, касающуюся файлов и

сетей. Система ввода-вывода Java характеризуется единством и непротиворечивостью. На самом деле, как только вы поймете ее основы, остальную часть системы ввода-вывода освоить несложно. Здесь предложен общий обзор ввода-вывода, а подробное описание приведено в главах 22 и 23.

Потоки данных

Ввод-вывод в программах на Java выполняется через потоки данных. *Поток данных* (stream) — это абстракция, которая либо производит, либо потребляет информацию. Поток связан с физическим устройством посредством системы ввода-вывода Java. Все потоки ведут себя одинаково, даже если фактические физические устройства, с которыми они связаны, различаются. Таким образом, одни и те же классы и методы ввода-вывода могут применяться к разным типам устройств, что означает возможность абстрагирования входного потока от множества различных типов ввода: из дискового файла, клавиатуры или сетевого сокета. Точно так же поток вывода может относиться к консоли, дисковому файлу или сетевому подключению. Потоки данных являются чистым способом работы с вводом-выводом, при котором в каждой части вашего кода не требуется учет отличий, например, между клавиатурой и сетью. Потоки данных Java реализованы внутри иерархий классов, определенных в пакете `java.io`.

На заметку! В дополнение к потоковому вводу-выводу, определенному в `java.io`, язык Java также предоставляет ввод-вывод на основе буферов и каналов, который определен в `java.nio` и его подчиненных пакетах. Ввод-вывод такого вида обсуждается в главе 23.

Потоки байтовых и символьных данных

В Java определены два типа потоков ввода-вывода: байтовые и символьные. *Потоки байтовых данных* предлагают удобные средства для обработки ввода и вывода байтов. Они используются, например, при чтении или записи двоичных данных. *Потоки символьных данных* предоставляют удобные средства для обработки ввода и вывода символов. Они применяют Unicode и, следовательно, допускают интернационализацию. Кроме того, в ряде случаев потоки символьных данных эффективнее потоков байтовых данных.

Первоначальная версия Java (Java 1.0) не включала потоки символьных данных, поэтому весь ввод-вывод был ориентирован на байты. Потоки символьных данных появились в версии Java 1.1, а некоторые классы и методы, ориентированные на байты, были объявлены нерекомендуемыми. Несмотря на то что унаследованный код, где потоки символьных данных не используются, встречается все реже, временами вы все еще можете с ним сталкиваться. Как правило, унаследованный код должен быть обновлен, чтобы надлежащим образом задействовать преимущества потоков символов.

Еще один момент: на самом низком уровне все операции ввода-вывода по-прежнему ориентированы на байты. Потоки символьных данных просто обеспечивают удобный и эффективный инструмент для обработки символов.

Обзор потоков данных, ориентированных на байты и на символы, представлен в последующих разделах.

Классы потоков байтовых данных

Потоки байтовых данных определяются с применением двух иерархий классов. Вверху находятся два абстрактных класса: `InputStream` и `OutputStream`. Каждый из них имеет несколько конкретных подклассов, которые справляются с отличиями между разными устройствами, такими как дисковые файлы, сетевые подключения и даже буферы памяти. Классы потоков байтовых данных из `java.io`, которые не объявлены нерекомендуемыми, показаны в табл. 13.1. Некоторые из этих классов обсуждаются далее в разделе, а другие описаны в части II книги. Помните, что для использования классов потоков должен быть импортирован пакет `java.io`.

Таблица 13.1. Классы потоков байтовых данных в `java.io`, которые не объявлены нерекомендуемыми

Класс потока данных	Описание
`BufferedInputStream`	Буферизованный поток ввода
`BufferedOutputStream`	Буферизованный поток вывода
`ByteArrayInputStream`	Поток ввода, который выполняет чтение из байтового массива
`ByteArrayOutputStream`	Поток вывода, который выполняет запись в байтовый массив
`DataInputStream`	Поток ввода, который содержит методы для чтения стандартных типов данных Java
`DataOutputStream`	Поток ввода, который содержит методы для записи стандартных типов данных Java
`FileInputStream`	Поток ввода, который выполняет чтение из файла
`FileOutputStream`	Поток вывода, который выполняет запись в файл
`FilterInputStream`	Реализует `InputStream`
`FilterOutputStream`	Реализует `OutputStream`
`InputStream`	Абстрактный класс, который описывает поток ввода
`ObjectInputStream`	Поток ввода для объектов
`ObjectOutputStream`	Поток вывода для объектов
`OutputStream`	Абстрактный класс, который описывает поток вывода
`PipedInputStream`	Канал ввода

Окончание табл. 13.1

Класс потока данных	Описание
PipedOutputStream	Канал вывода
PrintStream	Поток вывода, который содержит методы print() и println()
PushbackInputStream	Поток ввода, который позволяет возвращать байты в этот поток ввода
SequenceInputStream	Поток ввода, являющийся комбинацией двух и более потоков ввода, которые будут читаться последовательно друг за другом

В абстрактных классах InputStream и OutputStream определено несколько ключевых методов, реализуемых другими классами потоков. Двумя наиболее важными из них являются read() и write(), которые выполняют, соответственно, чтение и запись байтов данных. У каждого имеется абстрактная форма, которая должна быть переопределена в производных классах потоков.

Классы потоков символьных данных

Потоки символьных данных определяются с помощью двух иерархий классов. Вверху находятся два абстрактных класса: Reader и Writer. Они обрабатывают потоки символов Unicode. Для каждого из них в Java предусмотрено несколько конкретных подклассов. Классы потоков символьных данных в java.io описаны в табл. 13.2.

Таблица 13.2. Классы потоков символьных данных в java.io

Класс потока данных	Описание
BufferedReader	Буферизованный поток ввода символьных данных
BufferedWriter	Буферизованный поток вывода символьных данных
CharArrayReader	Поток ввода, который выполняет чтение из символьного массива
CharArrayWriter	Поток вывода, который выполняет запись в символьный массив
FileReader	Поток ввода, который выполняет чтение из файла
FileWriter	Поток вывода, который выполняет запись в файл
FilterReader	Фильтрующее средство чтения
FilterWriter	Фильтрующее средство записи
InputStreamReader	Поток ввода, который выполняет трансляцию байтов в символы

Окончание табл. 13.2

Класс потока данных	Описание
`LineNumberReader`	Поток ввода, который подсчитывает строки
`OutputStreamWriter`	Поток вывода, который выполняет трансляцию символов в байты
`PipedReader`	Канал ввода
`PipedWriter`	Канал вывода
`PrintWriter`	Поток вывода, который содержит методы `print()` и `println()`
`PushbackReader`	Поток ввода, который позволяет возвращать байты в этот поток ввода
`Reader`	Абстрактный класс, описывающий поток ввода символьных данных
`StringReader`	Поток ввода, который выполняет чтение из строки
`StringWriter`	Поток вывода, который выполняет запись в строку
`Writer`	Абстрактный класс, описывающий поток вывода символьных данных

В абстрактных классах `Reader` и `Writer` определено несколько ключевых методов, реализуемых другими классами потоков. Двумя наиболее важными методами считаются `read()` и `write()`, которые выполняют, соответственно, чтение и запись символов данных. У каждого есть абстрактная форма, которая должна переопределяться в производных классах потоков.

Предопределенные потоки данных

Как вам известно, все программы на Java автоматически импортируют пакет `java.lang`, в котором определен класс `System`, инкапсулирующий ряд аспектов исполняющей среды. Скажем, с помощью некоторых его методов можно получить текущее время и настройки разнообразных свойств, связанных с системой. Класс `System` также содержит три предопределенные потоковые переменные: `in`, `out` и `err`. Они объявлены в `System` как поля `public`, `static` и `final`, т.е. могут использоваться в любой другой части программы без привязки к конкретному объекту `System`.

Поле `System.out` ссылается на стандартный поток вывода. По умолчанию это консоль. Поле `System.in` ссылается на стандартный поток ввода, в качестве которого по умолчанию выступает клавиатура. Поле `System.err` ссылается на стандартный поток вывода ошибок, по умолчанию также являющийся консолью. Однако упомянутые потоки могут быть перенаправлены на любое совместимое устройство ввода-вывода.

System.in представляет собой объект типа InputStream, а System.out и System.err — объекты типа PrintStream. Это потоки байтовых данных, хотя обычно они применяются для чтения символов из консоли и записи символов на консоль. При желании вы можете поместить их внутрь потоков символьных данных.

В предыдущих главах в примерах использовалось поле System.out. Практически аналогично можно применять System.err. Как объясняется в следующем разделе, использование System.in чуть сложнее.

Чтение консольного ввода

На заре развития Java единственным способом выполнения консольного ввода было применение потока байтовых данных. В настоящее время использование потока байтовых данных для чтения консольного ввода по-прежнему часто приемлемо, например, в примерах программ. Тем не менее, в коммерческих приложениях для чтения консольного ввода предпочтительнее применять символьный поток, что облегчает интернационализацию и сопровождение программы.

Консольный ввод в Java выполняется (прямо или косвенно) путем чтения из System.in. Один из способов получения символьного потока, присоединенного к консоли, предусматривает помещение System.in в оболочку BufferedReader. Класс BufferedReader поддерживает буферизованный поток ввода. Ниже приведен часто используемый конструктор:

```
BufferedReader(Reader inputReader)
```

Здесь inputReader представляет собой поток, связанный с создаваемым экземпляром BufferedReader, а Reader — абстрактный класс. Одним из его конкретных подклассов является InputStreamReader, который преобразует байты в символы. Начиная с версии JDK 17, точный способ получения объекта InputStreamReader, связанного с System.in, изменился. В прошлом для этой цели обычно применялся следующий конструктор InputStreamReader:

```
InputStreamReader(InputStream inputStream)
```

Поскольку System.in ссылается на объект типа InputStream, его можно указывать в аргументе inputStream. Таким образом, приведенная далее строка кода демонстрирует ранее широко используемый подход к созданию объекта BufferedReader, подключенного к клавиатуре:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

После выполнения этого оператора переменная br становится символьным потоком, связанным с консолью через System.in.

Однако, начиная с JDK 17, при создании объекта InputStreamReader рекомендуется явно указывать набор символов, ассоциированный с консолью. *Набор символов* определяет способ сопоставления байтов с символами. Обычно, когда набор символов не задан, применяется стандартная кодировка JVM. Тем не менее, в случае консоли набор символов, используемый для кон-

сольного ввода, может отличаться от стандартного набора символов. Таким образом, теперь рекомендуется применять следующую форму конструктора InputStreamReader:

```
InputStreamReader(InputStream inputStream, Charset набор-символов)
```

В аргументе charset должен использоваться набор символов, ассоциированный с консолью, который возвращается с помощью charset() — нового метода, добавленного к классу Console в JDK 17 (см. главу 22). Объект Console получается вызовом метода System.console(), который возвращает ссылку на консоль или null, если консоль отсутствует. Следовательно, теперь показанная ниже кодовая последовательность демонстрирует один из способов помещения System.in в оболочку BufferedReader:

```
Console con = System.console();  // получить объект Console
if(con==null) return;            // возврат, если консоль отсутствует

BufferedReader br = new
  BufferedReader(new InputStreamReader(System.in, con.charset()));
```

Разумеется, в тех случаях, когда известно, что консоль будет присутствовать, последовательность можно сократить:

```
BufferedReader br = new
  BufferedReader(new InputStreamReader(System.in,
                                       System.console().charset()));
```

Поскольку для запуска примеров, рассматриваемых в книге, (очевидно) требуется консоль, мы будем применять именно такую форму.

Еще один момент: можно также получить объект Reader, который уже ассоциирован с консолью, с помощью метода reader(), определенного в Console. Однако мы будем использовать только что описанный подход с InputStreamReader, т.к. он явно иллюстрирует возможность взаимодействия потоков байтовых и символьных данных.

Чтение символов

Для чтения символа из BufferedReader предназначен метод read(). Вот версия read(), которая будет применяться:

```
int read() throws IOException
```

При каждом вызове метод read() читает символ из потока ввода и возвращает его в виде целочисленного значения. Он возвращает -1 при попытке чтения в конце потока. Как видите, он может сгенерировать исключение IOException.

В представленной далее программе демонстрируется работа метода read(), читающего символы с консоли до тех пор, пока пользователь не введет q. Обратите внимание, что любые исключения ввода-вывода, которые могут возникнуть, генерируются в main(). Такой подход является обычным при чтении из консоли в простых примерах программ вроде приведенных в этой книге, но в более сложных приложениях исключения можно обрабатывать явно.

```
// Использование объекта BufferedReader для чтения символов с консоли.
import java.io.*;

class BRRead {
  blic static void main(String[] args) throws IOException
  {
    char c;
    BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in, System.console().charset()));

    System.out.println("Вводите символы; для выхода введите q.");
    // Читать символы.
    do {
      c = (char) br.read();
      System.out.println(c);
    } while(c != 'q');
  }
}
```

Ниже показан пример вывода, полученного в результате запуска программы:

```
Вводите символы; для выхода введите q.
123abcq
1
2
3
a
b
c
q
```

Вывод может немного отличаться от того, что вы ожидали, потому что `System.in` по умолчанию буферизирует строки, т.е. никакие входные данные фактически не передаются программе до тех пор, пока не будет нажата клавиша <Enter>. Несложно догадаться, что это не делает метод `read()` особенно ценным для интерактивного ввода с консоли.

Чтение строк

Для чтения строки с клавиатуры предназначена версия метода `readLine()`, которая является членом класса `BufferedReader` со следующей общей формой:

```
String readLine() throws IOException
```

Как видите, метод `readLine()` возвращает объект `String`.

В приведенной далее программе демонстрируется использование объекта `BufferedReader` и метода `readLine()`; программа читает и отображает строки текста до тех пор, пока не будет введено слово `stop`:

```
// Чтение строки с консоли с применением BufferedReader.
import java.io.*;

class BRReadLines {
  public static void main(String[] args) throws IOException
  {
```

```
    // Создать объект BufferedReader, используя System.in.
    BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in, System.console().charset()));

    String str;
    System.out.println("Вводите строки текста.");
    System.out.println("Для завершения введите stop.");
    do {
      str = br.readLine();
      System.out.println(str);
    } while(!str.equals("stop"));
  }
}
```

В следующем примере строится крошечный текстовый редактор. В нем создается массив объектов `String`, после чего читаются строки текста с сохранением каждой строки в массиве. Чтение выполняется вплоть до 100 строк или до ввода слова `stop`. Для чтения из консоли применяется объект `BufferedReader`.

```
// Крошечный текстовый редактор.
import java.io.*;

class TinyEdit {
  public static void main(String[] args) throws IOException
  {
    // Создать объект BufferedReader, используя System.in.
    BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in, System.console().charset()));

    String[] str = new String[100];
    System.out.println("Вводите строки текста.");
    System.out.println("Для завершения введите stop.");
    for(int i=0; i<100; i++) {
      str[i] = br.readLine();
      if(str[i].equals("stop")) break;
    }
    System.out.println("\nВот то, что вы ввели:");
    // Отобразить строки.
    for(int i=0; i<100; i++) {
      if(str[i].equals("stop")) break;
      System.out.println(str[i]);
    }
  }
}
```

Ниже приведен пример вывода, полученного в результате запуска программы:

```
Вводите строки текста.
Для завершения введите stop.
Первая строка.
Вторая строка.
Язык Java облегчает работу со строками.
Просто создайте объекты String.
stop
```

```
Вот то, что вы ввели:
Первая строка.
Вторая строка.
Язык Java облегчает работу со строками.
Просто создайте объекты String.
```

Запись консольного вывода

Консольный вывод проще всего обеспечить с помощью описанных ранее методов `print()` и `println()`, которые используются в большинстве примеров книги. Упомянутые методы определены в классе `PrintStream` (тип объекта, на который ссылается `System.out`). Несмотря на то что `System.out` является потоком байтовых данных, применять его для простого вывода в программе по-прежнему приемлемо. Тем не менее, в следующем разделе описана альтернатива на основе символов.

Поскольку класс `PrintStream` представляет собой выходной поток, производный от `OutputStream`, он также реализует низкоуровневый метод `write()`. Таким образом, `write()` можно использовать для записи в консоль. Вот простейшая форма `write()`, определенная в `PrintStream`:

```
void write(int byteval)
```

Метод `write()` записывает байт, указанный в аргументе `byteval`. Хотя аргумент `byteval` объявлен как `int`, записываются только младшие восемь битов. Ниже показан краткий пример, в котором `write()` применяется для вывода на экран символа "A", а за ним символа новой строки:

```
// Демонстрация использования System.out.write().
class WriteDemo {
  public static void main(String[] args) {
    int b;

    b = 'A';
    System.out.write(b);
    System.out.write('\n');
  }
}
```

Для вывода на консоль метод `write()` будет использоваться нечасто (несмотря на его удобство в ряде ситуаций), т.к. значительно проще применять методы `print()` и `println()`.

Класс `PrintWriter`

Хотя использовать `System.out` для записи в консоль вполне допустимо, вероятно, его лучше всего применять для целей отладки или в примерах программ, подобных тем, которые можно найти в этой книге. В реальных программах на Java рекомендуется осуществлять запись в консоль посредством потока `PrintWriter` — одного из символьных классов. Использование символьного класса для консольного вывода упрощает интернационализацию программы.

В классе PrintWriter определено несколько конструкторов; один из них, с которым мы будем иметь дело, показан ниже:

```
PrintWriter(OutputStream outputStream, boolean flushingOn)
```

Здесь outputStream является объектом типа OutputStream, а flushingOn определяет, будет ли поток вывода очищаться при каждом вызове метода println() (среди прочих). Если в аргументе flushingOn указано значение true, тогда очистка выполняется автоматически, а если false, то очистка автоматически не происходит.

Класс PrintWriter поддерживает методы print() и println(). Таким образом, эти методы можно применять тем же способом, как они использовались с System.out. Если аргумент не относится к простому типу, тогда методы PrintWriter вызывают метод toString() объекта и затем отображают результат.

Чтобы выполнить запись в консоль с применением PrintWriter, укажите System.out для потока вывода и автоматической очистки. Скажем, следующая строка кода создает объект PrintWriter, подключенный к консольному выводу:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

В показанном далее приложении иллюстрируется использование объекта PrintWriter для обработки консольного вывода:

```
// Демонстрация применения PrintWriter.
import java.io.*;

public class PrintWriterDemo {
  public static void main(String[] args) {
    PrintWriter pw = new PrintWriter(System.out, true);

    pw.println("Тестовая строка");
    int i = -7;
    pw.println(i);
    double d = 4.5e-7;
    pw.println(d);
  }
}
```

Вот вывод, генерируемый программой:

```
Тестовая строка
-7
4.5E-7
```

Помните, что нет ничего плохого в том, чтобы применять System.out для вывода простого текста на консоль, когда вы изучаете Java или отлаживаете свои программы. Однако использование PrintWriter облегчает интернационализацию реальных приложений. Из-за того, что применение PrintWriter в примерах программ, предложенных в книге, не дает никаких преимуществ, мы будем продолжать использовать System.out для выполнения записи в консоль.

Чтение файлов и запись в файлы

В Java предлагается несколько классов и методов, которые позволяют осуществлять чтение файлов и запись в файлы. Прежде чем мы начнем, важно отметить, что тема файлового ввода-вывода довольно обширна и подробно исследуется в части II. Цель настоящего раздела в том, чтобы представить основные методы чтения и записи в файл. Несмотря на применение потоки байтовых данных, такие приемы могут быть адаптированы к символьным потокам.

Двумя наиболее часто используемыми классами потоков являются `FileInputStream` и `FileOutputStream`, которые создают потоки байтовых данных, связанные с файлами. Для открытия файла нужно просто создать объект одного из этих классов, указывая имя файла в качестве аргумента конструктора. Хотя оба класса поддерживают дополнительные конструкторы, мы будем применять следующие формы:

```
FileInputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName) throws FileNotFoundException
```

В аргументе `fileName` указывается имя файла, который необходимо открыть. Если при создании входного потока файл не существует, тогда генерируется исключение `FileNotFoundException`. В случае потока вывода исключение `FileNotFoundException` генерируется, если файл не может быть открыт или создан. Исключение `FileNotFoundException` является подклассом `IOException`. Когда выходной файл открывается, любой ранее существовавший файл с таким же именем уничтожается.

На заметку! В ситуациях, когда присутствует диспетчер безопасности, несколько классов файлов, в числе которых `FileInputStream` и `FileOutputStream`, будут генерировать исключение `SecurityException`, если при попытке открытия файла произойдет нарушение безопасности. По умолчанию приложения, запускаемые через `java`, не используют диспетчер безопасности. По этой причине в примерах ввода-вывода, приведенных в книге, нет необходимости отслеживать возможное исключение `SecurityException`. Тем не менее, другие типы приложений могут работать с диспетчером безопасности, и файловый ввод-вывод, выполняемый таким приложением, способен генерировать исключение `SecurityException`. В таком случае вам придется соответствующим образом обработать данное исключение. Имейте в виду, что в версии JDK 17 диспетчер безопасности объявлен устаревшим и подлежащим удалению.

Завершив работу с файлом, вы должны его закрыть, что делается вызовом метода `close()`, который реализован как в `FileInputStream`, так и в `FileOutputStream`:

```
void close() throws IOException
```

Закрытие файла приводит к освобождению системных ресурсов, выделенных для файла, что позволяет их задействовать другим файлом. Из-за отказа от закрытия файла могут возникнуть "утечки памяти", причиной которых являются оставшиеся выделенными неиспользуемые ресурсы.

На заметку! Метод `close()` определен в интерфейсе `AutoCloseable` внутри пакета `java.lang`. Интерфейс `AutoCloseable` унаследован интерфейсом `Closeable` в `java.io`. Оба интерфейса реализуются классами потоков, в том числе `FileInputStream` и `FileOutputStream`.

Прежде чем двигаться дальше, важно отметить, что существуют два основных подхода, которые можно применять для закрытия файла по завершении работы с ним. Первый — традиционный подход, при котором метод `close()` вызывается явно, когда файл больше не нужен. Такой подход использовался во всех версиях Java перед выходом JDK 7 и потому встречается во всем унаследованном коде до JDK 7. Второй подход предусматривает применение появившегося в JDK 7 оператора `try` с ресурсами, который автоматически закрывает файл, когда в нем отпадает необходимость. При этом подходе явный вызов `close()` не производится. Поскольку вы все еще можете столкнуться со унаследованным кодом, написанным до JDK 7, важно знать и понимать традиционный подход. Кроме того, в некоторых ситуациях традиционный подход все еще может оказаться наилучшим и потому имеет смысл начать именно с него. Автоматизированный подход описан в следующем разделе.

Для чтения из файла можно использовать версию `read()`, определенную в `FileInputStream`, которая показана ниже:

```
int read() throws IOException
```

Каждый раз, когда метод `read()` вызывается, он читает один байт из файла и возвращает его в виде целочисленного значения. При попытке чтения в конце потока `read()` возвращает `-1` и также может сгенерировать исключение `IOException`.

В приведенной далее программе метод `read()` применяется для ввода и отображения содержимого файла, содержащего текст ASCII. Имя файла указывается в качестве аргумента командной строки.

```
/* Отображение содержимого текстового файла.
   Для использования программы укажите имя файла, который хотите просмотреть.
   Например, чтобы увидеть содержимое файла по имени TEST.TXT,
   введите следующую командную строку:

   java ShowFile TEST.TXT
*/

import java.io.*;

class ShowFile {
  public static void main(String[] args)
  {
    int i;
    FileInputStream fin;

    // Удостовериться, что имя файла было указано.
    if(args.length != 1) {
      System.out.println("Использование: ShowFile имя-файла");
      return;
    }

    // Попытаться открыть файл.
```

```
    try {
      fin = new FileInputStream(args[0]);
    } catch(FileNotFoundException e) {
      System.out.println("Не удалось открыть файл.");
      return;
    }
    // В данной точке файл открыт и может быть прочитан.
    // Следующий код читает символы до тех пор, пока не встретится EOF.
    try {
      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);
    } catch(IOException e) {
      System.out.println("Ошибка при чтении файла.");
    }
    // Закрыть файл.
    try {
      fin.close();
    } catch(IOException e) {
      System.out.println("Ошибка при закрытии файла.");
    }
  }
}
```

Обратите внимание в программе на блоки `try/catch`, которые обрабатывают возможные ошибки ввода-вывода. Каждая операция ввода-вывода отслеживается на предмет наличия исключений, и если исключение все же возникает, то оно обрабатывается. Имейте в виду, что в простых программах или примерах кода исключения ввода-вывода обычно просто генерируются в методе `main()`, как делалось в более ранних примерах консольного ввода-вывода. Вдобавок в реальном коде иногда удобно позволить исключению распространяться в вызывающую процедуру, тем самым сообщая вызывающей стороне о том, что операция ввода-вывода не удалась. Однако в большинстве примеров файлового ввода-вывода, рассматриваемых в книге, все исключения ввода-вывода обрабатываются явно, как показано в целях иллюстрации.

Хотя в предыдущем примере файловый поток закрывается после того, как файл был прочитан, существует вариант, который часто бывает полезным. Вариант предусматривает вызов метода `close()` внутри блока `finally`. При таком подходе все методы доступа к файлу содержатся в блоке `try`, а блок `finally` используется для закрытия файла. В итоге независимо от того, каким образом завершится блок `try`, файл будет закрыт. Вот как можно переделать блок `try` из предыдущего примера, который читает файл:

```
try {
  do {
    i = fin.read();
    if(i != -1) System.out.print((char) i);
  } while(i != -1);
} catch(IOException e) {
  System.out.println("Ошибка при чтении файла.");
```

```
} finally {
  // Закрыть файл при выходе из блока try.
  try {
    fin.close();
  } catch(IOException e) {
    System.out.println("Ошибка при закрытии файла.");
  }
}
```

Хотя в данном случае это не проблема, одно преимущество такого подхода в целом связано с тем, что если код, который получает доступ к файлу, завершается из-за какого-либо исключения, не связанного с вводом-выводом, то файл все равно закрывается в блоке `finally`.

Временами проще поместить те части программы, которые открывают файл и получают к нему доступ, в один блок `try` (вместо их разнесения), а затем применить блок `finally` для закрытия файла. Например, вот еще один способ написания программы `ShowFile`:

```
/* Отображение содержимого текстового файла.
   Для использования программы укажите имя файла, который хотите просмотреть.
   Например, чтобы увидеть содержимое файла по имени TEST.TXT,
   введите следующую командную строку:

   java ShowFile TEST.TXT

   В этом варианте код, который открывает и получает доступ к файлу,
   помещен в один блок try. Файл закрывается в блоке finally.
*/

import java.io.*;

class ShowFile {
  public static void main(String[] args)
  {
    int i;
    FileInputStream fin = null;

    // Удостовериться, что имя файла было указано.
    if(args.length != 1) {
      System.out.println("Использование: ShowFile имя-файла");
      return;
    }

    // Следующий код открывает файл, читает символы до тех пор, пока
    // не встретится EOF, и затем закрывает файл через блок finally.
    try {
      fin = new FileInputStream(args[0]);

      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);
    } catch(FileNotFoundException e) {
      System.out.println("Файл не найден.");
    } catch(IOException e) {
        System.out.println("Возникла ошибка ввода-вывода.");
    } finally {
```

```
      // Закрыть файл во всех случаях.
      try {
        if(fin != null) fin.close();
      } catch(IOException e) {
        System.out.println("Ошибка при закрытии файла.");
      }
    }
  }
}
```

Обратите внимание, что при таком подходе переменная `fin` инициализируется значением `null`. В блоке `finally` файл закрывается только в том случае, если значение `fin` не равно `null`. Прием работает, потому что `fin` не будет `null`, только если файл успешно открыт. Таким образом, метод `close()` не вызывается, если при открытии файла возникло исключение.

Последовательность `try/catch` в предыдущем примере можно сделать чуть более компактной. Поскольку исключение `FileNotFoundException` является подклассом `IOException`, его не нужно перехватывать отдельно. Например, вот последовательность, переписанная с целью устранения перехвата `FileNotFoundException`. В данном случае отображается стандартное сообщение об исключении, описывающее ошибку.

```
try {
  fin = new FileInputStream(args[0]);

  do {
    i = fin.read();
    if(i != -1) System.out.print((char) i);
  } while(i != -1);
} catch(IOException e) {
  System.out.println("Ошибка ввода-вывода: " + e);
} finally {
  // Закрыть файл во всех случаях.
  try {
    if(fin != null) fin.close();
  } catch(IOException e) {
    System.out.println("Ошибка при закрытии файла.");
  }
}
```

При таком подходе любая ошибка, в том числе ошибка, связанная с открытием файла, просто обрабатывается одним оператором `catch`. Из-за своей компактности этот подход используется во многих примерах ввода-вывода, представленных в книге. Тем не менее, имейте в виду, что такой подход не подходит в ситуациях, когда вы хотите отдельно обрабатывать отказ при открытии файла, такой как в случае, если пользователь неправильно набрал имя файла. В ситуации подобного рода вы можете запросить правильное имя, скажем, перед входом в блок `try`, где производится доступ к файлу.

Для выполнения записи в файл можно применять метод `write()`, определенный в `FileOutputStream`. Ниже показана его простейшая форма:

```
void write(int byteval) throws IOException
```

Метод `write()` записывает в файл байт, указанный в аргументе `byteval`. Хотя аргумент `byteval` объявлен как `int`, записываются только младшие восемь битов. Если во время записи возникает ошибка, тогда генерируется исключение `IOException`. В следующем примере метод `write()` используется для копирования файла:

```
/* Копирование файла.
   Для использования программы укажите имена исходного и целевого файлов.
   Например, чтобы скопировать файл по имени FIRST.TXT в файл по имени
   SECOND.TXT, введите следующую командную строку:

   java CopyFile FIRST.TXT SECOND.TXT
*/

import java.io.*;

class CopyFile {
  public static void main(String[] args) throws IOException
  {
    int i;
    FileInputStream fin = null;
    FileOutputStream fout = null;

    // Удостовериться, что были указаны оба файла.
    if(args.length != 2) {
     System.out.println("Использование: CopyFile исходный-файл целевой-файл");
     return;
    }

    // Копировать файл.
    try {
      // Попытаться открыть файлы.
      fin = new FileInputStream(args[0]);
      fout = new FileOutputStream(args[1]);

      do {
        i = fin.read();
        if(i != -1) fout.write(i);
      } while(i != -1);
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    } finally {
      try {
        if(fin != null) fin.close();
      } catch(IOException e2) {
        System.out.println("Ошибка при закрытии исходного файла.");
      }

      try {
        if(fout != null) fout.close();
      } catch(IOException e2) {
        System.out.println("Ошибка при закрытии целевого файла.");
      }
    }
  }
}
```

Обратите внимание, что при закрытии файлов в программе применяются два отдельных блока try, гарантируя закрытие обоих файлов, даже если вызов fin.close() сгенерирует исключение.

Следует отметить, что в двух предыдущих программах все потенциальные ошибки ввода-вывода обрабатывались с помощью исключений. Это отличается от ряда других языков программирования, где для сообщения об ошибках, связанных с файлами, используются коды ошибок. Исключения не только делают обработку файлов яснее, но и позволяют легко отличать состояние конца файла от файловых ошибок при выполнении ввода.

Автоматическое закрытие файла

В предыдущем разделе внутри примеров программ явно вызывался метод close() для закрытия файла, когда в нем исчезала необходимость. Как уже упоминалось, именно так файлы закрывались в версиях Java до JDK 7. Хотя этот подход по-прежнему актуален и полезен, в JDK 7 появилось средство, которое предлагает другой способ управления ресурсами, такими как файловые потоки, путем автоматизации процесса закрытия. Средство, иногда называемое *автоматическим управлением ресурсами* (automatic resource management — ARM), основано на расширенной версии оператора try. Главное преимущество автоматического управления ресурсами связано с тем, что оно предотвращает ситуации, в которых файл (или другой ресурс) по невнимательности не освобождается после того, как он больше не нужен. Ранее уже объяснялось, что игнорирование закрытия файла может привести к утечкам памяти и прочим проблемам.

Автоматическое управление ресурсами основано на расширенной форме оператора try:

```
try (спецификация-ресурса) {
  // использовать ресурс
}
```

Как правило, спецификация ресурса представляет собой оператор, который объявляет и инициализирует ресурс, скажем, файловый поток. Он состоит из объявления переменной, в котором переменная инициализируется ссылкой на управляемый объект. Когда блок try заканчивается, ресурс автоматически освобождается. В случае файла это означает автоматическое закрытие файла. (Соответственно нет необходимости явно вызывать метод close().) Конечно, такая форма try может также включать конструкции catch и finally. Она называется оператором try *с ресурсами.*

На заметку! Начиная с JDK 9, спецификация ресурса в try также может состоять из переменной, которая была объявлена и инициализирована ранее в программе. Однако эта переменная должна быть *фактически финальной*, т.е. после предоставления начального значения новое значение ей не присваивалось.

Оператор try с ресурсами можно применять только с теми ресурсами, которые реализуют интерфейс AutoCloseable, упакованный в java.lang. В интерфейсе AutoCloseable определен метод close(). Вдобавок интерфейс AutoCloseable унаследован интерфейсом Closeable в java.io. Оба интерфейса реализованы классами потоков. Таким образом, try с ресурсами можно использовать при работе с потоками, в том числе с файловыми потоками.

В качестве первого примера автоматического закрытия файла взгляните на переработанную версию программы ShowFile:

```
/* В этой версии программы ShowFile используется оператор try с ресурсами
   для автоматического закрытия файла после того, как он больше не нужен.
*/

import java.io.*;
class ShowFile {
  public static void main(String[] args)
  {
    int i;

    // Удостовериться, что имя файла было указано.
    if(args.length != 1) {
      System.out.println("Использование: ShowFile имя-файла");
      return;
    }

    // В следующем коде применяется оператор try с ресурсами для открытия
    // файла и затем его закрытия при покидании блока try.
    try(FileInputStream fin = new FileInputStream(args[0])) {
      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);
    } catch(FileNotFoundException e) {
      System.out.println("Файл не найден.");
    } catch(IOException e) {
      System.out.println("Произошла ошибка ввода-вывода.");
    }
  }
}
```

Обратите в программе особое внимание на то, каким образом файл открывается внутри оператора try:

```
try(FileInputStream fin = new FileInputStream(args[0])) {
```

Легко заметить, что в части спецификации ресурса оператора try объявляется объект FileInputStream по имени fin, которому затем присваивается ссылка на файл, открытый его конструктором. Таким образом, в данной версии программы переменная fin является локальной для блока try и создается при входе в него. Когда блок try заканчивается, поток, связанный с fin, автоматически закрывается неявным вызовом close(). Явно вызывать метод close() не понадобится, а потому не беспокойтесь о том, что вы забудете закрыть файл. В этом и состоит ключевое преимущество применения оператора try с ресурсами.

Важно понимать, что ресурс, объявленный в операторе try, неявно является final, т.е. присваивать ему ресурс после его создания нельзя. Кроме того, область действия ресурса ограничена оператором try с ресурсами.

Прежде чем двигаться дальше, полезно упомянуть о том, что начиная с JDK 10, при указании типа ресурса, объявленного в операторе try с ресурсами, можно использовать средство выведения типов локальных переменных. Для этого необходимо задать тип var, в результате чего тип ресурса будет выведен из его инициализатора. Например, оператор try в предыдущей программе теперь можно записать так:

```
try(var fin = new FileInputStream(args[0])) {
```

Здесь для переменной fin выводится тип FileInputStream, потому что именно к такому типу принадлежит его инициализатор. Поскольку многие читатели имеют дело со средами Java, предшествующими JDK 10, в операторах try с ресурсами в оставшихся главах книги выведение типов применяться не будет, чтобы код мог работать для максимально возможного числа читателей. Разумеется, в будущем вы должны рассмотреть возможность использования выведения типов в собственном коде.

В одиночном операторе try вы можете управлять более чем одним ресурсом. Для этого просто отделяйте спецификации ресурсов друг от друга точками с запятой. Ниже представлен пример, где показанная ранее программа CopyFile переделана с целью применения одного оператора try с ресурсами для управления fin и fout.

```
/* Версия CopyFile, в которой используется оператор try с ресурсами.
   Здесь демонстрируется управление двумя ресурсами (в данном случае
   файлами) с помощью одного оператора try.
*/

import java.io.*;

class CopyFile {
  public static void main(String[] args) throws IOException
  {
    int i;

    // Удостовериться, что были указаны оба файла.
    if(args.length != 2) {
     System.out.println("Использование: CopyFile исходный-файл целевой-файл");
     return;
    }

    // Открыть и управлять двумя файлами посредством оператора try.
    try (FileInputStream fin = new FileInputStream(args[0]);
         FileOutputStream fout = new FileOutputStream(args[1]))
    {

      do {
        i = fin.read();
        if(i != -1) fout.write(i);
      } while(i != -1);

    } catch(IOException e) {
```

```
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

Обратите внимание на способ открытия в программе исходного и целевого файлов внутри блока `try`:

```
try (FileInputStream fin = new FileInputStream(args[0]);
     FileOutputStream fout = new FileOutputStream(args[1]))
{
  // ...
```

По окончании такого блока `try` файлы `fin` и `fout` будут закрыты. Сравнив эту версию программы с предыдущей версией, вы заметите, что она намного короче. Возможность оптимизации исходного кода является дополнительным преимуществом автоматического управления ресурсами.

Существует еще один аспект оператора `try` с ресурсами, о котором следует упомянуть. В общем случае при выполнении блока `try` возможна ситуация, когда исключение внутри `try` приведет к возникновению другого исключения во время закрытия ресурса в конструкции `finally`. В случае "нормального" оператора `try` исходное исключение утрачивается, будучи вытесненным вторым исключением. Тем не менее, при использовании оператора `try` с ресурсами второе исключение *подавляется*, но не утрачивается. Взамен оно добавляется в список подавленных исключений, ассоциированных с первым исключением. Список подавленных исключений можно получить с помощью метода `getSuppressed()`, определенного в классе `Throwable`.

Из-за преимуществ, которые предлагает оператор `try` с ресурсами, он будет применяться во многих, хотя и не во всех примерах программ в настоящем издании книги. В некоторых примерах по-прежнему используется традиционный подход к закрытию ресурса. На то есть несколько причин. Во-первых, вы можете столкнуться с унаследованным кодом, который основан на традиционном подходе. Важно, чтобы все программисты на Java полностью разбирались в традиционном подходе при сопровождении унаследованного кода и чувствовали себя комфортно. Во-вторых, возможно, что некоторые программисты в течение какого-то времени продолжат работу в среде, предшествующей JDK 7. В таких ситуациях расширенная форма `try` не будет доступной. Наконец, в-третьих, могут быть случаи, когда явное закрытие ресурса более целесообразно, нежели автоматический подход. По перечисленным причинам в ряде примеров, приводимых в книге, все еще применятся традиционный подход с явным вызовом метода `close()`. Эти примеры не только иллюстрируют традиционную методику, но могут быть скомпилированы и запущены всеми читателями во всех средах.

Помните! В нескольких примерах используется традиционный подход к закрытию файлов как средство иллюстрации данного приема, который часто встречается в унаследованном коде. Однако в новом коде обычно лучше применять автоматизированный подход, поддерживаемый описанным ранее оператором `try` с ресурсами.

Модификаторы transient и volatile

В Java определены два интересных модификатора типов: transient и volatile. Такие модификаторы служат для обработки нескольких специализированных ситуаций.

Когда переменная экземпляра объявлена как transient, ее значение не должно предохраняться при сохранении объекта, например:

```
class T {
  transient int a;        // не предохраняется
  int b;                  // предохраняется
}
```

Если объект типа T записывается в область постоянного хранения, тогда содержимое a не будет сохранено, но сохранится содержимое b.

Модификатор volatile уведомляет компилятор о том, что снабженная им переменная может быть неожиданно изменена другими частями программы. Одна из таких ситуаций связана с многопоточными программами. В многопоточной программе иногда одна и та же переменная совместно используется двумя и более потоками. Из соображений эффективности каждый поток может хранить собственную закрытую копию такой общей переменной. Настоящая (или *главная*) копия переменной обновляется в разное время, скажем, при входе в синхронизированный метод. Хотя подход подобного рода работает нормально, нередко он может оказаться неэффективным. В некоторых случаях действительно имеет значение лишь тот факт, что главная копия переменной всегда отражает ее текущее состояние. Чтобы убедиться в этом, просто укажите переменную как volatile, тем самым сообщив компилятору о необходимости использования главной копии переменной volatile (или, по крайней мере, о поддержке любых закрытых копий в актуальном состоянии с главной копией и наоборот). Кроме того, доступ к общей переменной должен производиться в точном соответствии с порядком, которого требует программа.

Введение в instanceof

Иногда полезно знать тип объекта во время выполнения. Например, у вас может быть один поток выполнения, генерирующий объекты различных типов, и другой поток, который эти объекты обрабатывает. В такой ситуации может быть удобно, чтобы обрабатывающий поток знал типы всех объектов при их получении. Еще одна ситуация, когда важно знать тип объекта во время выполнения, связана с приведением. В Java недопустимое приведение становится причиной ошибки времени выполнения. Многие недопустимые приведения могут быть обнаружены на этапе компиляции. Тем не менее, приведения, вовлекающие иерархию классов, способны порождать недопустимые приведения, которые можно обнаружить только во время выполнения. Например, суперкласс по имени A может производить два подкласса с именами B и C. Таким образом, приведение объекта B к типу A или приведение объекта C к

типу A допустимо, но приведение объекта B к типу C (или наоборот) не является законным. Учитывая, что объект типа A может ссылаться на объекты либо B, либо C, то как узнать во время выполнения, на какой тип объекта фактически осуществляется ссылка, прежде чем пытаться привести его к типу C? Им может быть объект типа A, B или C. Если это объект типа B, тогда сгенерируется исключение времени выполнения. Язык Java предлагает операцию времени выполнения `instanceof`, позволяющую ответить на такой вопрос.

Первым делом необходимо отметить, что операция `instanceof` в версии JDK 17 была значительно улучшена благодаря новому мощному средству, основанному на сопоставлении с образцом. В текущей главе обсуждается традиционная форма операции `instanceof`, а ее расширенная форма рассматривается в главе 17.

Ниже показана общая форма традиционной операции `instanceof`:

```
objref instanceof type
```

Здесь `objref` представляет собой ссылку на экземпляр класса, а `type` — тип класса. Если аргумент `objref` относится к указанному типу или может быть к нему приведен, то результатом вычисления операции `instanceof` является `true`. В противном случае результатом будет `false`. Таким образом, `instanceof` — это инструмент, с помощью которого программа может получать информацию о типе объекта во время выполнения.

Работа операции `instanceof` демонстрируется в следующей программе:

```
// Демонстрация работы операции instanceof.
class A {
  int i, j;
}

class B {
  int i, j;
}

class C extends A {
  int k;
}

class D extends A {
  int k;
}

class InstanceOf {
  public static void main(String[] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    D d = new D();
    if(a instanceof A)
      System.out.println("a является экземпляром A");
    if(b instanceof B)
      System.out.println("b является экземпляром B");
    if(c instanceof C)
      System.out.println("c является экземпляром C");
```

```
    if(c instanceof A)
      System.out.println("c является экземпляром A");

    if(a instanceof C)
      System.out.println("a можно привести к C");

    System.out.println();

    // Сравнить типы производных классов.
    A ob;

    ob = d;  // ссылка на d
    System.out.println("ob теперь ссылается на d");
    if(ob instanceof D)
      System.out.println("ob является экземпляром D");

    System.out.println();

    ob = c;  // ссылка на c
    System.out.println("ob теперь ссылается на c");

    if(ob instanceof D)
      System.out.println("ob можно привести к D");
    else
      System.out.println("ob нельзя привести к D");

    if(ob instanceof A)
      System.out.println("ob можно привести к A");

    System.out.println();

    // Все объекты могут быть приведены к Object.
    if(a instanceof Object)
      System.out.println("a можно привести к Object");
    if(b instanceof Object)
      System.out.println("b можно привести к Object");
    if(c instanceof Object)
      System.out.println("c можно привести к Object");
    if(d instanceof Object)
      System.out.println("d можно привести к Object");
  }
}
```

Вот вывод, выдаваемый программой:

```
a является экземпляром A
b является экземпляром B
c является экземпляром C
c можно привести к A

ob теперь ссылается на d
ob является экземпляром D

ob теперь ссылается на c
ob нельзя привести к D
ob можно привести к A

a можно привести к Object
b можно привести к Object
c можно привести к Object
d можно привести к Object
```

В большинстве простых программ операция instanceof не нужна, потому что часто тип объекта, с которым приходится работать, известен. Однако операция instanceof может быть очень полезной для обобщенных подпрограмм, которые оперируют с объектами из сложной иерархии классов или созданы в коде, находящемся за пределами вашего прямого контроля. Как вы увидите, усовершенствования, касающиеся сопоставления с образцом, которые описаны в главе 17, упрощают применение instanceof.

Модификатор strictfp

С выходом Java 2 несколько лет назад модель вычислений с плавающей точкой была слегка смягчена. В частности, новая модель не требовала усечения ряда промежуточных значений, возникающих во время вычислений. В некоторых случаях это предотвращало переполнение или потерю значимости. За счет добавления к классу, методу или интерфейсу модификатора strictfp можно гарантировать, что вычисления с плавающей точкой (и, следовательно, все усечения) будут выполняться точно так же, как в более ранних версиях Java. В случае модификации класса с помощью strictfp все методы в классе тоже автоматически снабжаются модификатором strictfp. Тем не менее, начиная с JDK 17, все вычисления с плавающей точкой являются строгими, а модификатор strictfp устарел и больше не обязателен. Его использование теперь приводит к выдаче предупреждающего сообщения.

В приведенном далее примере иллюстрируется применение strictfp для версий Java, предшествующих JDK 17. В нем компилятор Java уведомляется о том, что во всех методах, определенных в MyClass, должна использоваться первоначальная модель вычислений с плавающей точкой:

```
strictfp class MyClass {  // ...
```

Откровенно говоря, большинству программистов никогда не приходилось применять модификатор strictfp, потому что он воздействовал лишь на крайне небольшую группу проблем.

Помните! Начиная с JDK 17, модификатор stricfp был объявлен устаревшим и теперь его использование приводит к выдаче предупреждающего сообщения.

Собственные методы

Временами, хотя и редко, может понадобиться вызвать подпрограмму, которая написана не на языке Java. Обычно такая подпрограмма существует в виде исполняемого кода для процессора и рабочей среды, т.е. является собственным кодом. Например, необходимость в вызове подпрограммы собственного кода зачастую возникает из-за желания сократить время выполнения. Или же может потребоваться задействовать специализированную стороннюю библиотеку, такую как пакет статистических расчетов. Однако поскольку программы на Java компилируются в байт-код, который затем

интерпретируется (или компилируется на лету) исполняющей средой Java, вызов подпрограммы собственного кода из программы на Java выглядит невозможным. К счастью, такой вывод неверен. Язык Java предлагает ключевое слово native, которое применяется для объявления методов собственного кода. После объявления эти методы можно вызывать в программе на Java подобно любым другим методам Java.

Чтобы объявить собственный метод, поместите перед ним модификатор native, но не определяйте тело метода, например:

```
public native int meth();
```

После объявления собственного метода вы должны написать собственный метод и выполнить довольно сложную последовательность шагов для его связывания с вашим кодом на Java. За текущими подробностями обращайтесь в документацию по Java.

Использование assert

Еще одним интересным ключевым словом является assert. Оно применяется на стадии разработки программы для создания *утверждения* — условия, которое должно быть истинным во время выполнения программы. Например, в программе может существовать метод, который всегда обязан возвращать положительное целочисленное значение. Вы можете проверить данный факт путем утверждения о том, что возвращаемое значение больше нуля, используя оператор assert. Если во время выполнения условие истинно, то никакие другие действия не предпринимаются. Тем не менее, если условие ложно, тогда генерируется объект AssertionError. Утверждения часто применяются на стадии тестирования, чтобы убедиться в том, что некоторое ожидаемое условие действительно удовлетворяется. В коде выпуска утверждения обычно не используются.

Ключевое слово assert имеет две формы. Вот первая из них:

```
assert condition;
```

Здесь condition — это выражение условия, результатом вычисления которого должно быть булевское значение. Если результат равен true, то утверждение истинно и никаких других действий не происходит. Если условие ложно, тогда утверждение терпит неудачу и генерируется стандартный объект AssertionError.

Ниже показана вторая форма assert:

```
assert condition: expr;
```

В данной версии выражение expr представляет собой значение, которое передается конструктору AssertionError. Это значение преобразуется в строковый формат и отображается в случае отказа утверждения. Как правило, в конструкции expr указывается строка, но допускается любое выражение не void, если оно определяет приемлемое строковое преобразование.

Далее приведен пример применения assert, в котором осуществляется проверка, что возвращаемое значение getnum() является положительным.

```
// Демонстрация использования assert.
class AssertDemo {
  static int val = 3;

  // Возвращает целое число.
  static int getnum() {
    return val--;
  }

  public static void main(String[] args)
  {
    int n;

    for(int i=0; i < 10; i++) {
      n = getnum();

      assert n > 0;  // потерпит неудачу, когда n равно 0

      System.out.println("n равно " + n);
    }
  }
}
```

Для включения проверки утверждений во время выполнения понадобится указать параметр -ea. Скажем, чтобы включить утверждения для AssertDemo, введите следующую строку:

```
java -ea AssertDemo
```

После компиляции и запуска программа выдаст такой вывод:

```
n равно 3
n равно 2
n равно 1
Exception in thread "main" java.lang.AssertionError
        at AssertDemo.main(AssertDemo.java:17)
Исключение в потоке main типа java.lang.AssertionError
        в AssertDemo.main(AssertDemo.java:17)
```

Внутри main() организованы многократные вызовы метода getnum(), который возвращает целочисленное значение. Возвращаемое значение getnum() присваивается переменной n и затем проверяется с помощью следующего оператора assert:

```
assert n > 0;  // потерпит неудачу, когда n равно 0
```

Такой оператор assert потерпит неудачу, когда n равно 0 (что произойдет после четвертого вызова), и сгенерируется исключение.

Ранее уже объяснялось, что разрешено указывать сообщение, которое будет отображаться в случае отказа утверждения. Например, если в предыдущей программе привести утверждение к следующему виду:

```
assert n > 0 : "n не является положительным!";
```

тогда сгенерируется показанный ниже вывод:

```
n равно 3
n равно 2
n равно 1
Exception in thread "main" java.lang.AssertionError: n не является
положительным! at AssertDemo.main(AssertDemo.java:17)
Исключение в потоке main типа java.lang.AssertionError: n не является
положительным! в AssertDemo.main(AssertDemo.java:17)
```

Один важный момент, связанный с утверждениями, который нужно понимать, касается того, что вы не должны полагаться на них при выполнении каких-либо действий, действительно требуемых программой. Причина в том, что обычно код выпуска будет выполняться с отключенными утверждениями. Например, взгляните на такой вариант предыдущей программы:

```
// Неудачный способ использования assert!!!
class AssertDemo {
  // Произвольное число.
  static int val = 3;

  // Возвращает целое число.
  static int getnum() {
    return val--;
  }

  public static void main(String[] args)
  {
    int n = 0;

    for(int i=0; i < 10; i++) {
      assert (n = getnum()) > 0;  // Поступать так не рекомендуется!

      System.out.println("n равно " + n);
    }
  }
}
```

В этой версии программы вызов `getnum()` перенесен внутрь оператора `assert`. Хотя код благополучно работает, когда утверждения включены, он будет функционировать некорректно при отключенных утверждениях, потому что вызов `getnum()` никогда не выполнится! На самом деле теперь переменную `n` необходимо инициализировать, т.к. компилятор идентифицирует, что присвоить значение `n` внутри оператора `assert` может не получиться.

Утверждения могут быть очень полезными, поскольку они упрощают вид проверки ошибок, распространенный на стадии разработки. Например, до появления `assert`, если в предыдущей программе вы хотели удостовериться в том, что значение `n` было положительным, пришлось бы использовать такую кодовую последовательность:

```
if(n < 0) {
  System.out.println("n является отрицательным!");
  return;  // или сгенерировать исключение
}
```

Проверка с применением `assert` требует только одной строки кода. Кроме того, удалять операторы `assert` из кода выпуска не понадобится.

Параметры включения и отключения проверки утверждений

При выполнении кода вы можете отключить проверку всех утверждений, используя параметр -da. Чтобы включить или отключить конкретный пакет (и все его подчиненные пакеты), необходимо указать его имя с тремя точками после опции -ea или -da. Скажем, для включения проверки утверждений в пакете MyPack применяйте следующую конструкцию:

```
-ea:MyPack...
```

Для отключения проверки утверждений в пакете MyPack используйте конструкцию:

```
-da:MyPack...
```

С помощью параметра -ea или -da можно также указывать класс. Например, вот как включить AssertDemo:

```
-ea:AssertDemo
```

Статическое импортирование

В состав языка Java входит средство *статического импортирования*, которое расширяет возможности ключевого слова import. За счет снабжения оператора import ключевым словом static этот оператор можно применять для импортирования статических членов класса или интерфейса. В случае использования статического импортирования к статическим членам можно получать доступ напрямую по их именам, не уточняя именем класса, что упрощает и сокращает синтаксис, требующийся для работы со статическим членом.

Чтобы понять полезность статического импортирования, давайте начнем с примера, где он не применяется. В следующей программе вычисляется гипотенуза прямоугольного треугольника. Она использует два статических метода из встроенного в Java класса Math, входящего в состав пакета java.lang. Первый метод — Math.pow(), который возвращает значение, возведенное в указанную степень, а второй — Math.sqrt(), возвращающий квадратный корень своего аргумента.

```
// Вычисление гипотенузы прямоугольного треугольника.
class Hypot {
  public static void main(String[] args) {
    double side1, side2;
    double hypot;
    side1 = 3.0;
    side2 = 4.0;

    // Обратите внимание на то, что sqrt() и pow() должны
    // быть уточнены именем их класса, т.е. Math.
    hypot = Math.sqrt(Math.pow(side1, 2) +
                      Math.pow(side2, 2));
```

```
    System.out.println("При заданных длинах сторон " +
                       side1 + " и " + side2 +
                      " гипотенуза равна " +
                       hypot);
  }
}
```

Поскольку `pow()` и `sqrt()` являются статическими методами, они должны вызываться с применением имени их класса `Math`, что приводит к несколько громоздкому вычислению гипотенузы:

```
hypot = Math.sqrt(Math.pow(side1, 2) +
                  Math.pow(side2, 2));
```

Как демонстрирует этот простой пример, необходимость указывать имя класса при каждом использовании `pow()` либо `sqrt()` (или любых других методов класса `Math`, таких как `sin()`, `cos()` и `tan()`) может стать утомительной.

В новой версии предыдущей программы показано, каким образом можно избавиться от надоедливого указания имени класса через статическое импортирование:

```
// Использование статического импортирования для помещения
// sqrt() и pow() в область видимости.
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

// Вычислить гипотенузу прямоугольного треугольника.
class Hypot {
  public static void main(String[] args) {
    double side1, side2;
    double hypot;

    side1 = 3.0;
    side2 = 4.0;

    // Методы sqrt() и pow() можно вызывать
    // сами по себе, без имени их класса.
    hypot = sqrt(pow(side1, 2) + pow(side2, 2));

    System.out.println("При заданных длинах сторон " +
                       side1 + " и " + side2 +
                       " гипотенуза равна " +
                       hypot);
  }
}
```

В приведенной выше версии имена `sqrt` и `pow` помещаются в область видимости с помощью следующих операторов статического импортирования:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

Такие операторы устраняют необходимость в уточнении `sqrt()` или `pow()` именем их класса. В итоге вычисление гипотенузы становится более удобным:

```
hypot = sqrt(pow(side1, 2) + pow(side2, 2));
```

Как видите, код этого вида значительно удобнее для восприятия.

Есть две основные формы оператора `import static`. Первая форма, которая применялась в предыдущем примере, помещает в область видимости одиночное имя и показана ниже:

```
import static pkg.type-name.static-member-name;
```

В конструкции `type-name` указывается имя класса или интерфейса, содержащего нужный статический член. Полное имя его пакета определяется конструкцией `pkg`. Имя статического члена указывается в конструкции `static-member-name`.

Вторая форма оператора `import static` импортирует все статические члены заданного класса или интерфейса и выглядит следующим образом:

```
import static pkg.type-name.*;
```

Если вы собираетесь использовать много статических методов или полей, определенных в классе, тогда такая форма позволит поместить их в область видимости, не указывая каждый метод или поле по отдельности. Скажем, в предыдущей программе можно было бы применять следующий единственный оператор `import`, чтобы поместить в область видимости методы `pow()` и `sqrt()` (наряду со всеми остальными статическими членами класса `Math`):

```
import static java.lang.Math.*;
```

Разумеется, статическое импортирование не ограничивается только классом `Math` или только методами. Например, приведенный ниже оператор `import static` помещает в область видимости статическое поле `System.out`:

```
import static java.lang.System.out;
```

После этого оператора можно выводить данные на консоль без указания `System`:

```
out.println("После импортирования System.out можно использовать out напрямую.");
```

Эффективность такого импортирования `System.out` является предметом споров. Хотя оно делает оператор короче, изучающим программу не сразу становится ясно, что под `out` понимается `System.out`.

Еще один момент: в дополнение к импортированию статических членов классов и интерфейсов, определенных в Java API, оператор `import static` также можно использовать для импортирования статических членов классов и интерфейсов, которые создаете вы сами.

Каким бы удобным ни было статическое импортирование, важно не злоупотреблять им. Не забывайте, что причиной организации библиотек Java в пакеты было стремление избежать конфликтов пространств имен. Импортирование статических членов приводит к их переносу в текущее пространство имен, тем самым увеличивая вероятность возникновения конфликтов между пространствами имен и непреднамеренного сокрытия имен. Если статический элемент используется в программе один или два раза, то лучше его не импортировать. К тому же некоторые статические имена, такие как `System.out`, настолько узнаваемы, что импортировать их не имеет особого смысла. Статическое им-

портирование предназначено для тех ситуаций, в которых статический член применяется многократно, скажем, при выполнении последовательности математических расчетов. В сущности, вы должны использовать средство статического импортирования, но не злоупотреблять им.

Вызов перегруженных конструкторов через this()

При работе с перегруженными конструкторами временами полезно, чтобы один конструктор вызывал другой. В языке Java для этого применяется другая форма ключевого слова this, общий вид которой показан ниже:

```
this(arg-list)
```

При выполнении this() первым запускается перегруженный конструктор со списком параметров, соответствующим тому, что указано в списке аргументов arg-list. Далее если внутри исходного конструктора имеются какие-либо операторы, то они выполняются. Вызов this() должен быть первым оператором внутри конструктора.

Чтобы лучше понять, как пользоваться this(), давайте рассмотрим небольшой пример. Для начала взгляните на следующий класс, где this() *не* применяется:

```
class MyClass {
  int a;
  int b;

  // Инициализировать a и b по отдельности.
  MyClass(int i, int j) {
    a = i;
    b = j;
  }

  // Инициализировать a и b одним и тем же значением.
  MyClass(int i) {
    a = i;
    b = i;
  }

  // Предоставить a и b стандартные значения 0.
  MyClass() {
    a = 0;
    b = 0;
  }
}
```

Класс MyClass содержит три конструктора, каждый из которых инициализирует поля a и b. Первому конструктору передаются индивидуальные значения для a и b. Второму передается только одно значение, которое присваивается как a, так и b. Третий предоставляет полям a и b стандартные значения, равные нулю. Используя this(), код MyClass можно переписать:

```
class MyClass {
  int a;
  int b;
```

```
  // Инициализировать a и b по отдельности.
  MyClass(int i, int j) {
    a = i;
    b = j;
  }
  // Инициализировать a и b одним и тем же значением.
  MyClass(int i) {
    this(i, i);  // вызывается MyClass(i, i)
  }
  // Предоставить a и b стандартные значения 0.
  MyClass() {
    this(0);     // вызывается MyClass(0)
  }
}
```

В этой версии `MyClass` единственным конструктором, который фактически присваивает значения полям `a` и `b`, является `MyClass(int, int)`. Два других конструктора просто прямо или косвенно вызывают конструктор `MyClass(int, int)` через `this()`. Например, давайте выясним, что происходит при выполнении следующего оператора:

```
MyClass mc = new MyClass(8);
```

Вызов `MyClass(8)` инициирует выполнение `this(8, 8)`, что транслируется в вызов `MyClass(8, 8)`, поскольку это версия конструктора `MyClass`, список параметров которой соответствует аргументам, переданным через `this()`. Теперь рассмотрим показанный ниже оператор, который задействует стандартный конструктор:

```
MyClass mc2 = new MyClass();
```

В данном случае вызывается `this(0)`, приводя к вызову `MyClass(0)`, потому что это конструктор с совпадающим списком параметров. Конечно, `MyClass(0)` затем вызывает `MyClass(0,0)`, как было описано выше.

Одна из причин удобства вызова перегруженных конструкторов через `this()` связана с возможностью предотвращения ненужного дублирования кода. Во многих ситуациях сокращение повторяющегося кода уменьшает время, необходимое для загрузки класса, т.к. зачастую объектный код оказывается короче. Это особенно важно для программ, доставляемых через Интернет, когда время загрузки является проблемой. Применение `this()` также содействует структурированию кода, когда конструкторы содержат большой объем дублированного кода.

Однако вы должны проявлять осторожность. Конструкторы, вызывающие `this()`, будут выполняться чуть медленнее, нежели те, в которые встроен весь код инициализации. Дело в том, что механизм вызова и возвращения, используемый при вызове второго конструктора, увеличивает накладные расходы. Если ваш класс будет применяться для создания лишь нескольких объектов или если конструкторы в классе, вызывающие `this()`, будут использоваться редко, то такое снижение производительности во время выполнения, вероятно, окажется незначительным. Но если ваш класс будет задействован

для создания большого количества объектов (порядка тысяч) во время выполнения программы, тогда негативное влияние увеличения накладных расходов может быть значительным. Поскольку создание объекта влияет на всех пользователей вашего класса, будут случаи, когда вам придется тщательно взвешивать преимущества более быстрой загрузки по сравнению с увеличением времени, необходимого для создания объекта.

Вот еще одно соображение: для очень коротких конструкторов вроде применяемых в классе `MyClass` размер объектного кода часто невелик, независимо от того, используется `this()` или нет. (В действительности бывают ситуации, когда размер объектного кода не уменьшается.) Это связано с тем, что байт-код, который настраивается и возвращается из вызова `this()`, добавляет инструкции в объектный файл. Следовательно, даже если дублированный код в подобных ситуациях устранен, то применение `this()` не обеспечит значительную экономию времени загрузки. Тем не менее, дополнительные затраты в виде накладных расходов на строительство каждого объекта все равно будут понесены. Таким образом, `this()` лучше всего применять к тем конструкторам, которые содержат большое количество кода инициализации, а не к конструкторам, просто устанавливающим значение нескольких полей.

Существуют два ограничения, которые необходимо учитывать при использовании `this()`.

Во-первых, в вызове `this()` нельзя применять переменную экземпляра класса конструктора. Во-вторых, в одном и том же конструкторе не разрешено использовать вызовы `super()` и `this()`, потому что каждый из них должен быть первым оператором в конструкторе.

Несколько слов о классах, основанных на значениях

Начиная с JDK 8, в состав Java входит концепция класса, *основанного на значении*, и несколько классов в Java API позиционируются как основанные на значениях. Классы, основанные на значениях, определяются согласно разнообразным правилам и ограничениям. Рассмотрим несколько примеров. Они должны быть финальными, как и их переменные экземпляра. Если метод `equals()` устанавливает, что два экземпляра класса, основанного на значениях, равны, то один экземпляр может применяться вместо другого. Кроме того, два одинаковых, но отдельно полученных экземпляра класса, основанного на значении, могут фактически быть одним и тем же объектом. Очень важно избегать использования экземпляров класса основанного на значении, для синхронизации. Применяются дополнительные правила и ограничения. Кроме того, определение классов, основанных на значениях, со временем несколько изменилось. Последние сведения о классах, основанных на значениях, а также о том, какие классы в библиотеке Java API помечены как основанные на значениях, можно узнать в документации по Java.

ГЛАВА 14 Обобщения

С момента выхода в 1995 году первоначальной версии 1.0 в Java появилось много новых средств. Одним из нововведений, оказавших глубокое и долгосрочное влияние, стали *обобщения*. Будучи введенными в JDK 5, обобщения изменили Java в двух важных аспектах. Во-первых, они добавили в язык новый синтаксический элемент. Во-вторых, они вызвали изменения во многих классах и методах основного API. На сегодняшний день обобщения являются неотъемлемой частью программирования на Java, а потому необходимо четко понимать это важное средство, которое подробно рассматривается в настоящей главе.

Благодаря использованию обобщений можно создавать классы, интерфейсы и методы, которые будут безопасно работать с разнообразными типами данных. Многие алгоритмы логически одинаковы независимо от того, к какому типу данных они применяются. Например, механизм, поддерживающий стек, будет одним и тем же, невзирая на то, элементы какого типа в нем хранятся — `Integer`, `String`, `Object` или `Thread`. С помощью обобщений можно определить алгоритм однократно и независимо от конкретного типа данных, после чего применять этот алгоритм к широкому спектру типов данных без каких-либо дополнительных усилий. Впечатляющие возможности добавленных в язык обобщений коренным образом изменили способ написания кода Java.

Одним из средств Java, на которое обобщения оказали наиболее значительное воздействие, вероятно, следует считать *инфраструктуру коллекций* (Collections Framework), которая входит в состав Java API и подробно обсуждается в главе 20, но полезно кратко упомянуть о ней уже сейчас. *Коллекция* представляет собой группу объектов. В инфраструктуре Collections Framework определено несколько классов вроде списков и карт, которые управляют коллекциями. Классы коллекций всегда были способны работать с любыми типами объектов. Обобщения обеспечили дополнительное преимущество — возможность использования классов коллекций с полной безопасностью в отношении типов. Таким образом, помимо того, что обобщения сами по себе являются мощным языковым элементом, они также позволяют

существенно улучшить существующее средство. Это еще одна причина, по которой обобщения стали настолько важным дополнением к Java.

В главе описан синтаксис, теория и методика применения обобщений. В ней также показано, каким образом обобщения поддерживают безопасность типов в некоторых ранее сложных случаях. После изучения текущей главы имеет смысл ознакомиться с главой 20, где рассматривается инфраструктура Collections Framework. Там вы найдете множество примеров обобщений в действии.

Что такое обобщения?

В своей основе термин *обобщения* означает *параметризованные типы*. Параметризованные типы важны, поскольку они позволяют создавать классы, интерфейсы и методы, где тип данных, с которым они работают, указывается в качестве параметра. Например, с использованием обобщений можно создать единственный класс, который автоматически работает с разными типами данных. Класс, интерфейс или метод, который оперирует на параметризованном типе, называется *обобщенным*.

Важно понимать, что язык Java всегда предоставлял возможность создавать обобщенные классы, интерфейсы и методы за счет оперирования ссылками типа `Object`. Поскольку `Object` является суперклассом для всех других классов, ссылка на `Object` способна ссылаться на объект любого типа. Таким образом, в исходном коде обобщенные классы, интерфейсы и методы при работе с различными типами объектов задействовали ссылки на `Object`. Проблема заключалась в том, что они не могли делать это с обеспечением безопасности в отношении типов.

Обобщения добавили недостающую безопасность в отношении типов. Они также упростили процесс, т.к. больше не нужно явно использовать приведения типов для преобразования между `Object` и типом данных, с которыми фактически выполняются операции. Благодаря обобщениям все приведения становятся автоматическими и неявными. В результате обобщения расширили возможности многократного применения кода с надлежащей безопасностью и легкостью.

Внимание! *Предостережение для программистов на языке C++*: хотя обобщения похожи на шаблоны в C++, они не совпадают. Между этими двумя подходами к обобщенным типам существует ряд фундаментальных отличий. Если у вас есть опыт написания кода на C++, тогда важно не делать поспешных выводов о том, как работают обобщения в Java.

Простой пример обобщения

Давайте начнем с простого примера обобщенного класса. В показанной ниже программе определены два класса — обобщенный класс `Gen` и класс `GenDemo`, использующий `Gen`.

```
// Простой обобщенный класс.
// Здесь T - параметр типа, который будет заменен
// реальным типом при создании объекта типа Gen.
class Gen<T> {
  T ob;  // объявить объект типа T
  // Передать конструктору ссылку на объект типа T.
  Gen(T o) {
    ob = o;
  }
  // Возвратить ob.
  T getOb() {
    return ob;
  }
  // Вывести тип T.
  void showType() {
    System.out.println("Типом T является " + ob.getClass().getName());
  }
}
// Демонстрация применения обобщенного класса.
class GenDemo {
  public static void main(String[] args) {
    // Создать объект Gen для объектов типа Integer.
    Gen<Integer> iOb;

    // Создать объект Gen<Integer> и присвоить ссылку на него
    // переменной iOb. Обратите внимание на использование автоупаковки
    // для инкапсуляции значения 88 внутри объекта Integer.
    iOb = new Gen<Integer>(88);

    // Вывести тип данных, используемых переменной iOb.
    iOb.showType();

    // Получить значение iOb. Обратите внимание,
    // что приведение не требуется.
    int v = iOb.getOb();
    System.out.println("значение: " + v);

    System.out.println();

    // Создать объект Gen для объектов типа String.
    Gen<String> strOb = new Gen<String> ("Тест с обобщениями");

    // Вывести тип данных, используемых переменной strOb.
    strOb.showType();

    // Получить значение strOb. Снова обратите внимание,
    // что приведение не требуется.
    String str = strOb.getOb();
    System.out.println("значение: " + str);
  }
}
```

Программа выдает следующий вывод:

```
Типом T является java.lang.Integer
значение: 88

Типом T является java.lang.String
значение: Тест с обобщениями
```

А теперь внимательно разберем программу.

Прежде всего, обратите внимание на объявление Gen:

```
class Gen<T> {
```

Здесь T — имя *параметра типа*. Оно применяется в качестве заполнителя для фактического типа, который будет передан конструктору Gen при создании объекта. Таким образом, T используется внутри Gen всякий раз, когда требуется параметр типа. Обратите внимание, что T содержится внутри угловых скобок <>. Такой синтаксис можно обобщить. Объявляемый параметр типа всегда указывается в угловых скобках. Поскольку Gen задействует параметр типа, Gen является обобщенным классом, который также называется *параметризованным типом*.

В объявлении Gen имя T не играет особой роли. Можно было бы применить любой допустимый идентификатор, но T используется по традиции. Кроме того, рекомендуется выбирать имена для параметров типов в виде односимвольных заглавных букв. Другими часто применяемыми именами параметров типов являются V и E. Еще один момент относительно имен параметров типов: начиная с JDK 10, использовать var в качестве имени параметра типа не разрешено.

Затем T применяется для объявления объекта ob:

```
T ob;  // объявить объект типа T
```

Как уже объяснялось, T — это заполнитель для фактического типа, который указывается при создании объекта Gen. Таким образом, ob будет объектом типа, переданного в T. Например, если в T передается тип String, тогда ob получит тип String.

Теперь взгляните, как выглядит конструктор класса Gen:

```
Gen(T o) {
  ob = o;
}
```

Обратите внимание, что его параметр o имеет тип T, т.е. фактический тип o определяется типом, переданным в T, когда создается объект Gen. Кроме того, поскольку и параметр o, и переменная-член ob относятся к типу T, при создании объекта Gen они будут иметь один и тот же фактический тип.

Параметр типа T также можно использовать для указания возвращаемого типа метода, как в случае показанного далее метода getOb():

```
T getOb() {
  return ob;
}
```

Из-за того, что ob также имеет тип T, его тип совместим с возвращаемым типом, заданным getOb().

Метод showType() выводит тип T, вызывая getName() на объекте Class, который возвращается вызовом getClass() на ob. Метод getClass() определен в Object и потому является членом всех типов классов. Он возвращает объект Class, соответствующий типу класса объекта, на котором вызы-

вается. В Class определен метод getName(), который возвращает строковое представление имени класса.

Работа с обобщенным классом Gen демонстрируется в классе GenDemo. Сначала создается версия Gen для целых чисел:

```
Gen<Integer> iOb;
```

Внимательно взгляните на приведенное выше объявление. Первым делом обратите внимание, что тип Integer в угловых скобках после Gen. В данном случае Integer — *аргумент типа*, который передается параметру типа Gen, т.е. T. Фактически создается версия Gen, в которой все ссылки на T транслируются в ссылки на Integer. Соответственно для такого объявления ob имеет тип Integer и возвращаемый тип getOb() имеет тип Integer.

Прежде чем двигаться дальше, важно отметить, что компилятор Java на самом деле не создает разные версии Gen или любого другого обобщенного класса. Хотя думать в таких терминах удобно, в действительности происходит иное — компилятор удаляет всю информацию об обобщенном типе, заменяя необходимые приведения, чтобы код *вел себя* так, как если бы создавалась конкретная версия Gen. Таким образом, в программе действительно существует только одна версия Gen. Процесс удаления информации об обобщенном типе называется *стиранием*, и позже в главе мы еще вернемся к этой теме.

В следующей строке кода переменной iOb присваивается ссылка на экземпляр версии класса Gen для Integer:

```
iOb = new Gen<Integer>(88);
```

Обратите внимание, что при вызове конструктора класса Gen также указывается аргумент типа Integer. Причина в том, что объект (в данном случае iOb), которому присваивается ссылка, имеет тип Gen<Integer>. Таким образом, ссылка, возвращаемая операцией new, тоже должна иметь тип Gen<Integer>. В противном случае возникнет ошибка на этапе компиляции. Скажем, показанное ниже присваивание вызовет ошибку на этапе компиляции:

```
iOb = new Gen<Double>(88.0);  // Ошибка!
```

Поскольку переменная iOb относится к типу Gen<Integer>, ее нельзя применять для ссылки на объект Gen<Double>. Эта проверка типов является одним из основных преимуществ обобщений, т.к. она обеспечивает безопасность типов.

На заметку! Позже в главе вы увидите, что синтаксис, используемый для создания экземпляра обобщенного класса, можно сократить. В интересах ясности будет применяться полный синтаксис.

Как было указано в комментариях внутри программы, присваивание задействует автоупаковку для инкапсуляции значения 88, представляющего собой число типа int, в объект Integer:

```
iOb = new Gen<Integer>(88);
```

Прием работает, потому что в классе Gen<Integer> определен конструктор, который принимает аргумент типа Integer. Так как ожидается объект Integer, компилятор Java автоматически поместит в него значение 88. Разумеется, присваивание можно было бы записать и явно:

```
iOb = new Gen<Integer>(Integer.valueOf(88));
```

Однако эта версия кода не принесет никакой пользы.

Далее программа выводит тип ob внутри iOb, т.е. Integer, и затем получает значение ob с помощью следующей строки:

```
int v = iOb.getOb();
```

Поскольку возвращаемым типом getOb() является T, замененный типом Integer при объявлении iOb, возвращаемым типом getOb() также оказывается Integer, который распаковывается в int, когда выполняется присваивание переменной v (типа int). Таким образом, нет нужды приводить возвращаемый тип getOb() к Integer. Конечно, использовать средство автораспаковки не обязательно. Предыдущую строку кода можно было бы записать и так:

```
int v = iOb.getOb().intValue();
```

Тем не менее, средство автораспаковки содействует компактности кода.

Затем в классе GenDemo объявляется объект типа Gen<String>:

```
Gen<String> strOb = new Gen<String>("Тест с обобщениями");
```

Поскольку аргументом типа является String, внутри Gen тип String подставляется вместо T, что (концептуально) создает версию Gen для String, как демонстрируют оставшиеся строки в программе.

Обобщения работают только со ссылочными типами

При объявлении экземпляра обобщенного типа передаваемый параметру типа аргумент типа должен быть ссылочным типом. Примитивный тип вроде int или char применять нельзя. Скажем, для экземпляра Gen передать в T можно любой тип класса, но передавать параметру типа какой-либо примитивный тип не разрешено. По этой причине следующее объявление будет незаконным:

```
Gen<int> intOb = new Gen<int>(53);  // Ошибка, нельзя использовать
                                    //   примитивный тип
```

Конечно, отсутствие возможности указать примитивный тип не считается серьезным ограничением, т.к. для инкапсуляции примитивного типа можно использовать оболочки типов (как было в предыдущем примере). Вдобавок механизм автоупаковки и автораспаковки в Java обеспечивает прозрачность работы с оболочками типов.

Обобщенные типы различаются на основе их аргументов типов

Ключевой момент, который необходимо понять относительно обобщенных типов, связан с тем, что ссылка на одну специфическую версию обобщенного типа несовместима по типу с другой версией того же обобщенного типа. Например, пусть только что показанная программа содержит строку кода с ошибкой и компилироваться не будет:

```
iOb = strOb;  // Ошибка!
```

Несмотря на то что и iOb, и strOb относятся к типу Gen<T>, они являются ссылками на разные типы, т.к. их аргументы типов отличаются. Таким способом обобщения обеспечивают безопасность типов и предотвращают ошибки.

Каким образом обобщения улучшают безопасность в отношении типов?

В этот момент вы можете задать себе следующий вопрос. Учитывая, что та же функциональность, что и в обобщенном классе Gen, может быть достигнута без обобщений, просто за счет указания Object в качестве типа данных и применения соответствующих приведений, то в чем польза от превращения Gen в обобщенный класс? Ответ заключается в том, что обобщения автоматически обеспечивают безопасность в отношении типов для всех операций с участием Gen. В процессе они избавляют вас от необходимости вводить приведения типов и проверять типы в коде вручную.

Чтобы понять преимущества обобщений, сначала рассмотрим показанную ниже программу, в которой создается необобщенный эквивалент класса Gen:

```
// Класс NonGen функционально эквивалентен Gen,
// но не задействует обобщения.
class NonGen {
  Object ob;  // ob теперь имеет тип Object

  // Передать конструктору ссылку на объект типа Object.
  NonGen(Object o) {
    ob = o;
  }

  // Возвратить объект типа Object.
  Object getOb() {
    return ob;
  }

  // Вывести тип ob.
  void showType() {
    System.out.println("Типом ob является " +
                       ob.getClass().getName());
  }
}

// Демонстрация применения необобщенного класса.
class NonGenDemo {
```

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

    // Создать экземпляр NonGen и сохранить в нем объект Integer.
    // Автоупаковка по-прежнему происходит.
    iOb = new NonGen(88);

    // Вывести тип данных, используемых переменной iOb.
    iOb.showType();

    // Получить значение iOb. На этот раз приведение обязательно.
    int v = (Integer) iOb.getOb();
    System.out.println("значение: " + v);

    System.out.println();

    // Создать еще один экземпляр NonGen и сохранить в нем объект String.
    NonGen strOb = new NonGen("Тест без обобщений");

    // Вывести тип данных, используемых переменной strOb.
    strOb.showType();

    // Получить значение strOb.
    // Снова обратите внимание, что необходимо приведение.
    String str = (String) strOb.getOb();
    System.out.println("значение: " + str);

    // Следующий код скомпилируется, но он концептуально ошибочен!
    iOb = strOb;
    v = (Integer) iOb.getOb();  // ошибка во время выполнения!
  }
}
```

С версией без обобщений связано несколько интересных моментов. Прежде всего, обратите внимание, что все случаи использования `T` в `NonGen` заменены типом `Object`, позволяя `NonGen` хранить объекты любого типа подобно обобщенной версии. Однако это также не позволяет компилятору Java получить реальное знание о типе данных, фактически хранящихся в `NonGen`, что плохо по двум причинам. Во-первых, для извлечения сохраненных данных необходимо применять явные приведения типов. Во-вторых, многие виды ошибок несоответствия типов невозможно обнаружить до стадии выполнения. Давайте проанализируем каждую проблему.

Взгляните на следующую строку:

```
int v = (Integer) iOb.getOb();
```

Поскольку возвращаемым типом `getOb()` является `Object`, необходимо приведение к `Integer`, чтобы значение автоматически распаковывалось и сохранялось в переменной `v`. Если вы удалите приведение, тогда программа не скомпилируется. В обобщенной версии такое приведение было неявным. В необобщенной версии приведение должно быть явным. Оно оказывается не только неудобством, но и потенциальным источником ошибки.

Теперь возьмем кодовую последовательность из конца программы:

```
// Следующий код скомпилируется, но он концептуально ошибочен!
iOb = strOb;
v = (Integer) iOb.getOb();  // ошибка во время выполнения!
```

Здесь strOb присваивается переменной iOb. Тем не менее, strOb ссылается на объект, который содержит строку, а не целое число. Такое присваивание синтаксически допустимо, потому что все ссылки типа NonGen одинаковы, и любая ссылка NonGen может ссылаться на любой другой объект NonGen. Однако семантически данное утверждение неверно, как показывает следующая строка. Здесь возвращаемый тип getOb() приводится к Integer, после чего делается попытка присвоить результат переменной v. Проблема в том, что iOb теперь ссылается на объект, который хранит String, а не Integer. К сожалению, без использования обобщений компилятор Java не может это узнать. Взамен при попытке приведения к Integer генерируется исключение во время выполнения. Как вам известно, возникновение исключений в коде во время выполнения считается крайне плохой практикой!

В случае применения обобщений описанная выше ситуация невозможна. Если бы такая последовательность действий была предпринята в версии программы с обобщениями, то компилятор перехватил бы ее и сообщил об ошибке, тем самым предотвратив серьезный сбой, который вызывает исключение во время выполнения. Возможность написания безопасного в отношении типов кода, где ошибки несоответствия типов перехватываются на этапе компиляции, является ключевым преимуществом обобщений. Хотя использовать ссылки Object для создания "обобщенного" кода можно было всегда, такой код не поддерживает безопасность к типам, а его неправильная эксплуатация зачастую приводит к возникновению исключений во время выполнения. Обобщения препятствуют этому. По сути, за счет обобщений ошибки во время выполнения преобразуются в ошибки на этапе компиляции, что считается крупным преимуществом.

Обобщенный класс с двумя параметрами типов

В обобщенном типе допускается объявлять больше, чем один параметр типа. Чтобы указать два или более параметров типов, просто применяйте список, разделенный запятыми. Например, следующий класс TwoGen является разновидностью класса Gen с двумя параметрами типов:

```
// Простой обобщенный класс с двумя параметрами типов: T и V.
class TwoGen<T, V> {
  T ob1;
  V ob2;

  // Передать конструктору ссылки на объекты типов T и V.
  TwoGen(T o1, V o2) {
    ob1 = o1;
    ob2 = o2;
  }

  // Вывести типы T и V.
  void showTypes() {
    System.out.println("Типом T является " +
                       ob1.getClass().getName());
```

```
    System.out.println("Типом V является " + ob2.getClass().getName());
  }

  T getOb1() {
    return ob1;
  }

  V getOb2() {
    return ob2;
  }
}

// Демонстрация использования TwoGen.
class SimpGen {
  public static void main(String[] args) {

    TwoGen<Integer, String> tgObj =
      new TwoGen<Integer, String>(88, "Обобщения");

    // Вывести типы.
    tgObj.showTypes();

    // Получить и вывести значения.
    int v = tgObj.getOb1();
    System.out.println("значение: " + v);

    String str = tgObj.getOb2();
    System.out.println("значение: " + str);
  }
}
```

Ниже показан вывод, генерируемый программой:

```
Типом T является java.lang.Integer
Типом V является java.lang.String
значение: 88
значение: Обобщения
```

Обратите внимание на объявление класса TwoGen:

```
class TwoGen<T, V> {
```

В классе TwoGen определены два параметра типов, T и V, разделенные запятой. Поскольку он имеет два параметра типов, при создании объекта конструктору TwoGen должны быть переданы два аргумента типов:

```
TwoGen<Integer, String> tgObj =
  new TwoGen<Integer, String>(88, "Обобщения");
```

В данном случае вместо T подставляется тип Integer, а вместо V — тип String.

Несмотря на отличие между двумя аргументами типов в рассмотренном примере, оба типа могут быть одинаковыми. Например, следующая строка кода вполне допустима:

```
TwoGen<String, String> x = new TwoGen<String, String> ("A", "B");
```

В этом случае T и V будут иметь тип String. Разумеется, если бы аргументы типов всегда были одинаковыми, то наличие двух параметров расценивалось бы как излишество.

Общая форма обобщенного класса

Синтаксис обобщений, показанный в предшествующих примерах, можно свести к общей форме. Вот синтаксис объявления обобщенного класса:

```
class class-name<type-param-list> {  // ...
```

В `class-name` указывается имя класса, а в `type-param-list` — список параметров типов.

А так выглядит полный синтаксис для объявления ссылки на обобщенный класс и создания экземпляра:

```
class-name<type-arg-list> var-name =
  new class-name<type-arg-list>(cons-arg-list);
```

Здесь `class-name` — имя класса, `type-arg-list` — список аргументов типов, `var-name` — имя переменной, `cons-arg-list` — список аргументов конструктора.

Ограниченные типы

В предшествующих примерах параметры типов можно было заменить любым типом класса, что подходит для многих целей, но иногда полезно ограничить типы, которые разрешено передавать параметру типа. Предположим, что необходимо создать обобщенный класс, содержащий метод, который возвращает среднее значение массива чисел. Кроме того, вы хотите использовать этот класс для получения среднего значения массива чисел любого типа, включая `int`, `float` и `double`. Таким образом, тип чисел желательно указывать обобщенно с применением параметра типа. Чтобы создать класс подобного рода, вы можете попробовать поступить примерно так:

```
// (Безуспешная) попытка создать обобщенный класс Stats,
// который мог бы вычислять среднее значение массива
// чисел любого заданного типа.
//
// Этот класс содержит ошибку!
class Stats<T> {
  T[] nums;  // nums - массив элементов типа T

  // Передать конструктору ссылку на массив типа T.
  Stats(T[] o) {
    nums = o;
  }

  // Во всех случаях возвращать результат типа double.
  double average() {
    double sum = 0.0;
    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue();  // Ошибка!!!

    return sum / nums.length;
  }
}
```

Метод average() класса Stats пытается получить версию double каждого числа в массиве nums, вызывая doubleValue(). Поскольку все числовые классы, такие как Integer и Double, являются подклассами Number, а в Number определен метод doubleValue(), то данный метод доступен всем числовым классам-оболочкам. Проблема связана с тем, что компилятор не может заранее знать, что вы собираетесь создавать объекты Stats, используя только числовые типы. В итоге при компиляции Stats выдается сообщение об ошибке, указывающее на то, что метод doubleValue() неизвестен. Для решения проблемы нужен какой-то способ уведомления компилятора о том, что в T планируется передавать только числовые типы. Вдобавок необходимо каким-то образом *гарантировать*, что действительно передаются *только* числовые типы.

Для обработки таких ситуаций в Java предусмотрены *ограниченные типы*. Когда определяется параметр типа, вы можете создать верхнюю границу, объявляющую суперкласс, от которого должны быть порождены все аргументы типов. Цель достигается за счет применения конструкции extends при указании параметра типа:

```
<T extends superclass>
```

Таким образом, тип T может быть заменен только суперклассом, указанным в superclass, или подклассами этого суперкласса. В результате superclass определяет включающий верхний предел.

Прием можно использовать для исправления показанного ранее класса Stats, указав Number в качестве верхней границы:

```
// В этой версии класса Stats аргументом типа для T должен быть
// либо Number, либо класс, производный от Number.
class Stats<T extends Number> {
  T[] nums;  // массив элементов класса Number или его подкласса
  // Передать конструктору ссылку на массив элементов
  // класса Number или его подкласса.
  Stats(T[] o) {
    nums = o;
  }

  // Во всех случаях возвращать результат типа double.
  double average() {
    double sum = 0.0;

    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue();

    return sum / nums.length;
  }
}

// Демонстрация использования Stats.
class BoundsDemo {
  public static void main(String[] args) {

    Integer[] inums = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new Stats<Integer>(inums);
    double v = iob.average();
    System.out.println("Среднее значение iob равно " + v);
```

```
    Double[] dnums = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Stats<Double> dob = new Stats<Double>(dnums);
    double w = dob.average();
    System.out.println("Среднее значение dob равно " + w);

    // Следующий код не скомпилируется, т.к. String
    // не является подклассом Number.
    // String[] strs = { "1", "2", "3", "4", "5" };
    // Stats<String> strob = new Stats<String>(strs);

    // double x = strob.average();
    // System.out.println("Среднее значение strob равно " + v);
  }
}
```

Вот вывод:

```
Среднее значение iob равно 3.0
Среднее значение dob равно 3.3
```

Обратите внимание, как теперь объявляется класс `Stats`:

```
class Stats<T extends Number> {
```

Поскольку теперь тип `T` ограничен классом `Number`, компилятору Java известно, что все объекты типа `T` могут вызывать метод `doubleValue()`, т.к. он объявлен в `Number`. Это и само по себе считается большим преимуществом. Тем не менее, в качестве дополнительного бонуса ограничение `T` также предотвращает создание нечисловых объектов `Stats`. Например, если вы удалите комментарии из строк в конце программы и затем попробуете скомпилировать код заново, то получите ошибки на этапе компиляции, потому что `String` не является подклассом `Number`.

Для определения границы кроме типа класса можно также применять тип интерфейса. На самом деле в качестве границ разрешено указывать несколько интерфейсов. Вдобавок граница может включать как тип класса, так и один или несколько интерфейсов. В таком случае тип класса должен быть указан первым. Когда граница содержит тип интерфейса, то допускаются только аргументы типов, реализующие этот интерфейс. При указании привязки, которая имеет класс и интерфейс или несколько интерфейсов, для их соединения используйте операцию `&`, что в итоге создает *пересечения типов*. Например:

```
class Gen<T extends MyClass & MyInterface> {  // ...
```

Здесь `T` ограничивается классом `MyClass` и интерфейсом `MyInterface`. Таким образом, любой аргумент типа, передаваемый типу `T`, должен быть подклассом `MyClass` и реализовывать `MyInterface`. Интересно отметить, что пересечение типов можно также применять в приведении.

Использование аргументов с подстановочными знаками

Какой бы полезной ни была безопасность к типам, иногда она может мешать совершенно приемлемым конструкциям. Скажем, имея класс `Stats`,

показанный в конце предыдущего раздела, предположим, что вы хотите добавить метод по имени isSameAvg(), который выясняет, содержат ли два объекта Stats массивы, дающие одно и то же среднее значение, независимо от типа числовых данных в каждом из них. Например, если в одном объекте хранятся значения 1.0, 2.0 и 3.0 типа double, а в другом — целочисленные значения 2, 1 и 3, тогда средние значения будут одинаковыми. Один из способов реализации метода isSameAvg() предусматривает передачу ему аргумента Stats с последующим сравнением среднего значения этого аргумента и вызывающего объекта с возвращением true, только если средние значения совпадают. Например, вы хотите иметь возможность вызывать isSameAvg(), как показано ниже:

```
Integer[] inums = { 1, 2, 3, 4, 5 };
Double[] dnums = { 1.1, 2.2, 3.3, 4.4, 5.5 };

Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double> dob = new Stats<Double>(dnums);

if(iob.isSameAvg(dob))
  System.out.println("Средние значения одинаковы.");
else
  System.out.println("Средние значения отличаются.");
```

На первый взгляд создание isSameAvg() выглядит простой задачей. Так как класс Stats является обобщенным, а его метод average() способен работать с любым типом объекта Stats, кажется, что создать isSameAvg() несложно. К сожалению, проблемы начинаются, как только вы пытаетесь объявить параметр типа Stats. Поскольку Stats — параметризованный тип, что вы укажете для параметра типа в Stats при объявлении параметра данного типа?

Поначалу вы можете подумать о решении, где T применяется в качестве параметра типа:

```
// Работать не будет!
// Выяснить, одинаковы ли два средних значения.
boolean isSameAvg(Stats<T> ob) {
  if(average() == ob.average())
    return true;

  return false;
}
```

Проблема предпринятой попытки в том, что такое решение будет работать только с другими объектами Stats, тип которых совпадает с типом вызывающего объекта. Скажем, если вызываемый объект относится к типу Stats<Integer>, то параметр ob тоже должен иметь тип Stats<Integer>. Его нельзя использовать, например, для сравнения среднего значения объекта типа Stats<Double> со средним значением объекта типа Stats<Short>. Следовательно, этот подход будет работать лишь в очень узком контексте и не даст общего (т.е. обобщенного) решения.

Чтобы создать обобщенный метод isSameAvg(), потребуется задействовать еще одну особенность обобщений Java: аргумент с *подстановочным* знаком. Аргумент с подстановочным знаком указывается посредством символа ? и представляет неизвестный тип. Вот как с его помощью можно записать метод isSameAvg():

```
// Выяснить, одинаковы ли два средних значения.
// Обратите внимание на использование подстановочного знака.
boolean isSameAvg(Stats<?> ob) {
  if(average() == ob.average())
    return true;

  return false;
}
```

Здесь Stats<?> соответствует любому объекту Stats, позволяя сравнивать средние значения любых двух объектов Stats, что и демонстрируется в следующей программе:

```
// Использование подстановочного знака.
class Stats<T extends Number> {
  T[] nums;  // массив элементов класса Number или его подкласса

  // Передать конструктору ссылку на массив элементов
  // класса Number или его подкласса.
  Stats(T[] o) {
    nums = o;
  }

  // Во всех случаях возвращать результат типа double.
  double average() {
    double sum = 0.0;

    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue();

    return sum / nums.length;
  }

  // Выяснить, одинаковы ли два средних значения.
  // Обратите внимание на использование подстановочного знака.
  boolean isSameAvg(Stats<?> ob) {
    if(average() == ob.average())
      return true;

    return false;
  }
}

// Демонстрация применения подстановочного знака.
class WildcardDemo {
  public static void main(String[] args) {
    Integer[] inums = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new Stats<Integer>(inums);
    double v = iob.average();
    System.out.println("Среднее значение iob равно " + v);

    Double[] dums = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Stats<Double> dob = new Stats<Double>(dnums);
```

```
    double w = dob.average();
    System.out.println("Среднее значение dob равно " + w);

    Float[] fnums = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F };
    Stats<Float> fob = new Stats<Float>(fnums);
    double x = fob.average();
    System.out.println("Среднее значение fob равно " + x);

    // Выяснить, какие массивы имеют одинаковые средние значения.
    System.out.print("Средние значения iob и dob ");
    if(iob.isSameAvg(dob))
      System.out.println("одинаковы.");
    else
      System.out.println("отличаются.");

    System.out.print("Средние значения iob и fob ");
    if(iob.isSameAvg(fob))
      System.out.println("одинаковы.");
    else
      System.out.println("отличаются.");
  }
}
```

Ниже приведен вывод:

```
Среднее значение iob равно 3.0
Среднее значение dob равно 3.3
Среднее значение fob равно 3.0
Средние значения iob и dob отличаются.
Средние значения iob и fob одинаковы.
```

И последнее замечание: важно понимать, что подстановочный знак не влияет на то, какой тип объектов `Stats` можно создавать — это регулируется конструкцией `extends` в объявлении `Stats`. Подстановочный знак просто соответствует любому *допустимому* объекту `Stats`.

Ограниченные аргументы с подстановочными знаками

Аргументы с подстановочными знаками могут быть ограничены во многом так же, как и параметр типа. Ограниченный аргумент с подстановочным знаком особенно важен при создании обобщенного типа, который будет оперировать на иерархии классов. Чтобы понять причину, давайте проработаем пример. Рассмотрим следующую иерархию классов, инкапсулирующих координаты:

```
// Двумерные координаты.
class TwoD {
  int x, y;

  TwoD(int a, int b) {
    x = a;
    y = b;
  }
}

// Трехмерные координаты.
```

```
class ThreeD extends TwoD {
  int z;

  ThreeD(int a, int b, int c) {
    super(a, b);
    z = c;
  }
}

// Четырехмерные координаты.
class FourD extends ThreeD {
  int t;

  FourD(int a, int b, int c, int d) {
    super(a, b, c);
    t = d;
  }
}
```

На вершине иерархии находится класс TwoD, который инкапсулирует двумерную координату X, Y. Класс ThreeD унаследован от класса TwoD и добавляет третье измерение, создавая координату X, Y, Z. Класс FourD унаследован от класса ThreeD и добавляет четвертое измерение (время), давая четырехмерную координату.

Далее представлен обобщенный класс Coords, в котором хранится массив координат:

```
// Этот класс хранит массив объектов координат.
class Coords<T extends TwoD> {
  T[] coords;

  Coords(T[] o) {
    coords = o;
  }
}
```

Обратите внимание, что в классе Coords указан параметр типа, ограниченный TwoD, т.е. любой массив, хранящийся в объекте Coords, будет содержать объекты класса TwoD или одного из его подклассов.

Теперь предположим, что вы хотите написать метод, который отображает координаты X и Y для каждого элемента в массиве coords объекта Coords. Поскольку все типы объектов Coords имеют как минимум две координаты (X и Y), это легко сделать с помощью подстановочного знака:

```
static void showXY(Coords<?> c) {
  System.out.println("Координаты X Y:");
  for(int i=0; i < c.coords.length; i++)
    System.out.println(c.coords[i].x + " " +
                       c.coords[i].y);
    System.out.println();
}
```

Из-за того, что Coords является ограниченным обобщенным типом, применяющим TwoD в качестве верхней границы, все объекты, которые можно использовать для создания объекта Coords, будут массивами элементов клас-

са TwoD или производных от него классов. Таким образом, метод showXY() способен отображать содержимое любого объекта Coords.

Но что, если вы хотите создать метод, отображающий координаты X, Y и Z объекта ThreeD или FourD? Проблема в том, что не все объекты Coords будут иметь три координаты, т.к. объект Coords<TwoD> располагает только X и Y. Как тогда написать метод, отображающий координаты X, Y и Z для объектов Coords<ThreeD> и Coords<FourD>, и одновременно предотвратить применение этого метода с объектами Coords<TwoD>? Ответ — воспользоваться *ограниченным аргументом с подстановочными знаками.*

В ограниченном аргументе с подстановочным знаком задается верхняя или нижняя граница для аргумента типа, что позволяет сузить диапазон типов объектов, с которыми будет работать метод. Наиболее распространенным ограничением является верхняя граница, создаваемая с помощью конструкции extends почти так же, как при создании ограниченного типа.

Используя ограниченный аргумент с подстановочным знаком, легко создать метод, который отображает координаты X, Y и Z объекта Coords, если он действительно имеет указанные три координаты. Например, следующий метод showXYZ() выводит координаты X, Y и Z элементов, хранящихся в объекте Coords, если эти элементы действительно относятся к типу ThreeD (или к типу, производному от ThreeD):

```
static void showXYZ(Coords<? extends ThreeD> c) {
  System.out.println("Координаты X Y Z:");
  for(int i=0; i < c.coords.length; i++)
    System.out.println(c.coords[i].x + " " +
                       c.coords[i].y + " " +
                       c.coords[i].z);
  System.out.println();
}
```

Обратите внимание, что к подстановочному знаку в объявлении параметра c добавлена конструкция extends. Она указывает, что ? может соответствовать любому типу, если он является ThreeD или классом, производным от ThreeD. Таким образом, конструкция extends устанавливает верхнюю границу того, чему может соответствовать подстановочный знак. Из-за такой привязки метод showXYZ() можно вызывать со ссылками на объекты типа Coords<ThreeD> или Coords<FourD>, но не со ссылкой на объект типа Coords<TwoD>. Попытка вызвать showXZY() со ссылкой на объект типа Coords<TwoD> приводит к ошибке на этапе компиляции, что обеспечивает безопасность в отношении типов.

Ниже показана полная программа, демонстрирующая действия ограниченного аргумента с подстановочным знаком:

```
// Ограниченный аргумент с подстановочным знаком.

// Двумерные координаты.
class TwoD {
  int x, y;

  TwoD(int a, int b) {
```

```
    x = a;
    y = b;
  }
}

// Трехмерные координаты.
class ThreeD extends TwoD {
  int z;

  ThreeD(int a, int b, int c) {
    super(a, b);
    z = c;
  }
}

// Четырехмерные координаты.
class FourD extends ThreeD {
  int t;

  FourD(int a, int b, int c, int d) {
    super(a, b, c);
    t = d;
  }
}

// Этот класс хранит массив объектов координат.
class Coords<T extends TwoD> {
  T[] coords;

  Coords(T[] o) {
    coords = o;
  }
}

// Демонстрация использования ограниченного
// аргумента с подстановочным знаком.
class BoundedWildcard {
  static void showXY(Coords<?> c) {
    System.out.println("Координаты X Y:");
    for(int i=0; i < c.coords.length; i++)
      System.out.println(c.coords[i].x + " " +
                         c.coords[i].y);
    System.out.println();
  }

  static void showXYZ(Coords<? extends ThreeD> c) {
    System.out.println("Координаты X Y Z:");
    for(int i=0; i < c.coords.length; i++)
      System.out.println(c.coords[i].x + " " +
                         c.coords[i].y + " " +
                         c.coords[i].z);
    System.out.println();
  }

  static void showAll(Coords<? extends FourD> c) {
    System.out.println("Координаты X Y Z T:");
    for(int i=0; i < c.coords.length; i++)
      System.out.println(c.coords[i].x + " " +
```

```
                    c.coords[i].y + " " +
                    c.coords[i].z + " " +
                    c.coords[i].t);
    System.out.println();
  }

  public static void main(String[] args) {
    TwoD[] td = {
      new TwoD(0, 0),
      new TwoD(7, 9),
      new TwoD(18, 4),
      new TwoD(-1, -23)
    };

    Coords<TwoD> tdlocs = new Coords<TwoD>(td);

    System.out.println("Содержимое tdlocs.");
    showXY(tdlocs);      // Нормально, это объект TwoD
    // showXYZ(tdlocs);  // Ошибка, не объект ThreeD
    // showAll(tdlocs);  // Ошибка, не объект FourD

    // Создать  несколько объектов FourD.
    FourD[] fd = {
      new FourD(1, 2, 3, 4),
      new FourD(6, 8, 14, 8),
      new FourD(22, 9, 4, 9),
      new FourD(3, -2, -23, 17)
    };

    Coords<FourD> fdlocs = new Coords<FourD>(fd);

    System.out.println("Содержимое fdlocs.");
    // Все вызовы выполнятся успешно.
    showXY(fdlocs);
    showXYZ(fdlocs);
    showAll(fdlocs);
  }
}
```

Вот вывод, генерируемый программой:

```
Содержимое tdlocs.
Координаты X Y:
0 0
7 9
18 4
-1 -23

Содержимое fdlocs.
Координаты X Y:
1 2
6 8
22 9
3 -2

Координаты X Y Z:
1 2 3
6 8 14
22 9 4
3 -2 -23
```

```
Координаты X Y Z T:
1 2 3 4
6 8 14 8
22 9 4 9
3 -2 -23 17
```

Обратите внимание на закомментированные строки:

```
// showXYZ(tdlocs);  // Ошибка, не объект ThreeD
// showAll(tdlocs);  // Ошибка, не объект FourD
```

Так как `tdlocs` представляет собой объект `Coords(TwoD)`, его нельзя применять для вызова `showXYZ()` или `showAll()`, потому что этому препятствуют ограниченные аргументы с подстановочными знаками в их объявлениях. Чтобы удостовериться в сказанном, попробуйте удалить символы комментария и затем скомпилировать программу. Вы получите ошибки на этапе компиляции из-за несоответствия типов.

В общем случае для установления верхней границы аргумента с подстановочным знаком используйте выражение следующего вида:

```
<? extends superclass>
```

где `superclass` — имя класса, служащего верхней границей. Не забывайте, что это включающая конструкция, поскольку класс, формирующий верхнюю границу (т.е. указанный в `superclass`), также находится в пределах границ.

Вы также можете указать нижнюю границу аргумента с подстановочным знаком, добавив к объявлению конструкцию `super` с таким общим видом:

```
<? super subclass>
```

В данном случае допустимыми аргументами будут только классы, являющиеся суперклассами `subclass`. Конструкция тоже включающая.

Создание обобщенного метода

Как было проиллюстрировано в предшествующих примерах, методы внутри обобщенного класса могут использовать параметр типа класса и, следовательно, автоматически становятся обобщенными по отношению к параметру типа. Однако можно объявить обобщенный метод с одним или несколькими собственными параметрами типов. Более того, можно создавать обобщенный метод внутри необобщенного класса.

Давайте начнем с примера. В показанной ниже программе объявляется необобщенный класс по имени `GenMethDemo`, а в этом классе определяется статический обобщенный метод по имени `isIn()`, который выясняет, является ли объект членом массива. Его можно применять с любым типом объекта и массива при условии, что массив содержит объекты, совместимые с типом искомого объекта.

```
// Демонстрация определения простого обобщенного метода.
class GenMethDemo {

  // Выяснить, присутствует ли объект в массиве.
  static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y) {
```

```
    for(int i=0; i < y.length; i++)
      if(x.equals(y[i])) return true;
    return false;
  }

  public static void main(String[] args) {
    // Использовать isIn() для объектов Integer.
    Integer[] nums = { 1, 2, 3, 4, 5 };
    if(isIn(2, nums))
      System.out.println("2 присутствует в nums");
    if(!isIn(7, nums))
      System.out.println("7 не присутствует в nums");
    System.out.println();
    // Использовать isIn() для объектов String.
    String[] strs = { "one", "two", "three",
                      "four", "five" };
    if(isIn("two", strs))
      System.out.println("two присутствует в strs");
    if(!isIn("seven", strs))
      System.out.println("seven не присутствует в strs");
    // Не скомпилируется! Типы должны быть совместимыми.
    // if(isIn("two", nums))
    //   System.out.println("two присутствует в nums");
  }
}
```

Программа генерирует следующий вывод:

```
2 присутствует в nums
7 не присутствует в nums

two присутствует в strs
seven не присутствует в strs
```

Теперь займемся исследованием метода `isIn()`. Взгляните на его объявление:

```
static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y) {
```

Параметры типа объявляются *перед* возвращаемым типом метода. Кроме того, обратите внимание на конструкцию `T extends Comparable<T>`. Интерфейс `Comparable` объявлен в пакете `java.lang`. Класс, реализующий `Comparable`, определяет объекты, которые можно упорядочивать. Таким образом, требование верхней границы как `Comparable` гарантирует, что метод `isIn()` может использоваться только с объектами, которые обладают способностью участвовать в сравнениях. Интерфейс `Comparable` является обобщенным, и его параметр типа указывает тип сравниваемых объектов. (Вскоре вы увидите, как создавать обобщенный интерфейс.) Далее обратите внимание, что тип `V` ограничен сверху типом `T`. Соответственно тип `V` должен быть либо тем же самым, что и тип `T`, либо подклассом `T`. Такое отношение гарантирует, что метод `isIn()` можно вызывать только с аргументами, которые совмести-

мы друг с другом. Вдобавок метод `isIn()` определен как статический, что позволяет вызывать его независимо от любого объекта. Тем не менее, важно понимать, что обобщенные методы могут быть как статическими, так и нестатическими. В этом смысле нет никаких ограничений.

Метод `isIn()` вызывается внутри `main()` с применением обычного синтаксиса вызова без необходимости указывать аргументы типа. Дело в том, что типы аргументов распознаются автоматически, а типы `T` и `V` надлежащим образом корректируются. Например, при первом вызове типом первого аргумента оказывается `Integer` (из-за автоупаковки), что приводит к замене `T` на `Integer`:

```
if(isIn(2, nums))
```

Базовым типом второго аргумента тоже является `Integer` и потому `Integer` становится заменой для `V`. Во втором вызове используются типы `String`, так что типы `T` и `V` заменяются на `String`.

Хотя выведения типов будет достаточно для большинства вызовов обобщенных методов, при необходимости аргумент типа можно указывать явно. Скажем, вот как выглядит первый вызов `isIn()`, когда указаны аргументы типов:

```
GenMethDemo.<Integer, Integer>isIn(2, nums)
```

Конечно, в данном случае указание аргументов типов не дает какого-либо выигрыша. Кроме того, в JDK 8 выведение типов было улучшено в том, что касается методов. В результате в настоящее время встречается меньше случаев, когда требуются явные аргументы типов.

Теперь обратите внимание на закомментированный код:

```
// if(isIn("two", nums))
//   System.out.println("two присутствует в nums");
```

Удалив комментарии и попытавшись скомпилировать программу, вы получите ошибку. Причина в том, что параметр типа `V` ограничен `T` с помощью конструкции `extends` в объявлении `V`, т.е. тип `V` должен быть либо `T`, либо подклассом `T`. В рассматриваемой ситуации первый аргумент имеет тип `String`, что превращает `T` в `String`, а второй — тип `Integer`, который не является подклассом `String`. В итоге возникает ошибка несоответствия типов на этапе компиляции. Такая способность обеспечения безопасности к типам представляет собой одно из самых важных преимуществ обобщенных методов.

Синтаксис, применяемый для создания `isIn()`, можно обобщить. Вот синтаксис обобщенного метода:

```
<type-param-list > ret-type meth-name (param-list) { // ...
<список-параметров-типов> возвращаемый-тип имя-метода (список-параметров)
{  // ...
```

Во всех случаях в `type-param-list` указывается список параметров типов, разделенных запятыми. Кроме того, `ret-type` — это возвращаемый тип, `meth-name` — имя метода и `param-list` — список параметров. В обобщенном методе список параметров типов предшествует возвращаемому типу.

Обобщенные конструкторы

Конструкторы могут быть обобщенными, даже когда их класс таковым не является. Например, взгляните на следующую короткую программу:

```
// Использование обобщенного конструктора.
class GenCons {
  private double val;

  <Textends Number> GenCons(T arg) {
    val = arg.doubleValue();
  }

  void showVal() {
    System.out.println("val: " + val);
  }
}

class GenConsDemo {
  public static void main(String[] args) {

    GenCons test = new GenCons(100);
    GenCons test2 = new GenCons(123.5F);

    test.showVal();
    test2.showVal();
  }
}
```

Вот вывод:

```
val: 100.0
val: 123.5
```

Поскольку в `GenCons()` указан параметр обобщенного типа, который обязан быть подклассом `Number`, конструктор `GenCons()` можно вызывать с любым числовым типом, включая `Integer`, `Float` и `Double`. Следовательно, хотя `GenCons` — не обобщенный классом, его конструктор является обобщенным.

Обобщенные интерфейсы

В дополнение к обобщенным классам и методам также могут существовать обобщенные интерфейсы, которые определяются аналогично обобщенным классам. Ниже приведен пример, где создается интерфейс `MinMax` с методами `min()` и `max()`, которые должны возвращать минимальное и максимальное значения в наборе объектов.

```
// Пример обобщенного интерфейса.

// Интерфейс для нахождения минимального и максимального значений.
interface MinMax<T extends Comparable<T>> {
  T min();
  T max();
}

// Реализовать интерфейс MinMax.
class MyClass<T extends Comparable<T>> implements MinMax<T> {
  T[] vals;
```

```
  MyClass(T[] o) {
    vals = o;
  }

  // Возвратить минимальное значение в vals.
  public T min() {
    T v = vals[0];

    for(int i=1; i < vals.length; i++)
      if(vals[i].compareTo(v) < 0)
        v = vals[i];

    return v;
  }

  // Возвратить максимальное значение в vals.
  public T max() {
    T v = vals[0];

    for(int i=1; i < vals.length; i++)
      if(vals[i].compareTo(v) > 0)
        v = vals[i];
    return v;
  }
}

class GenIFDemo {
  public static void main(String[] args) {
    Integer[] inums = {3, 6, 2, 8, 6 };
    Character[] chs = {'b', 'r', 'p', 'w' };

    MyClass<Integer> iob = new MyClass<Integer>(inums);
    MyClass<Character> cob = new MyClass<Character>(chs);

    System.out.println("Максимальное значение в inums: " + iob.max());
    System.out.println("Минимальное значение в inums: " + iob.min());

    System.out.println("Максимальное значение в chs: " + cob.max());
    System.out.println("Минимальное значение в chs: " + cob.min());
  }
}
```

Программа генерирует следующий вывод:

```
Максимальное значение в inums: 8
Минимальное значение в inums: 2
Максимальное значение в chs: w
Минимальное значение в chs: b
```

Хотя понимание большинства аспектов в программе не должно вызывать затруднений, необходимо отметить пару ключевых моментов. Прежде всего, обратите внимание на способ объявления интерфейса `MinMax`:

```
interface MinMax<T extends Comparable<T>> {
```

Обобщенный интерфейс объявляется подобно обобщенному классу. В данном случае указан параметр типа `T` с верхней границей `Comparable`. Как объяснялось ранее, `Comparable` — это интерфейс, определенный в `java.lang`, который задает способ сравнения объектов. Его параметр типа указывает тип сравниваемых объектов.

Затем интерфейс MinMax реализуется классом MyClass. Взгляните на объявление MyClass:

```
class MyClass<T extends Comparable<T>> implements MinMax<T> {
```

Обратите особое внимание на то, как параметр типа T объявляется в MyClass и далее передается классу MinMax. Поскольку для MinMax требуется тип, который реализует Comparable, реализующий класс (в данном случае MyClass) должен указывать ту же самую границу. Более того, после установления этой границы нет никакой необходимости указывать ее снова в конструкции implements. На самом деле поступать так было бы неправильно. Например, приведенный ниже код некорректен и потому не скомпилируется:

```
// Ошибка!
class MyClass<T extends Comparable<T>>
  implements MinMax<T extends Comparable<T>> {
```

Установленный параметр типа просто передается интерфейсу без дальнейших изменений.

Вообще говоря, если класс реализует обобщенный интерфейс, то этот класс тоже должен быть обобщенным, по крайней мере, принимая параметр типа, который передается интерфейсу. Скажем, следующая попытка объявления MyClass ошибочна:

```
class MyClass implements MinMax<T> {  // Ошибка!
```

Поскольку в MyClass не объявляет параметр типа, то передать его в MinMax невозможно. В таком случае идентификатор T попросту неизвестен и компилятор сообщит об ошибке. Разумеется, если класс реализует *специфический тип* обобщенного интерфейса, тогда реализующий класс не обязан быть обобщенным:

```
class MyClass implements MinMax<Integer> {  // Нормально
```

Обобщенный интерфейс предлагает два преимущества. Во-первых, его можно реализовать для разных типов данных. Во-вторых, он позволяет накладывать ограничения (т.е. границы) на типы данных, для которых может быть реализован интерфейс. В примере с MinMax в T могут передаваться только типы, реализующие интерфейс Comparable.

Вот общий синтаксис обобщенного интерфейса:

```
interface interface-name<type-param-list> {  // ...
```

В interface-name указывается имя интерфейса, а в type-param-list — список параметров типов, разделенных запятыми. При реализации обобщенного интерфейса классом class-name в type-arg-list необходимо указывать аргументы типов:

```
class class-name<type-param-list> implements interface-name<type-arg-list> {
```

Низкоуровневые типы и унаследованный код

Поскольку до выхода JDK 5 поддержка обобщений отсутствовала, необходимо было обеспечить какой-то путь перехода от старого кода, предшествующего обобщениям. Кроме того, такой путь перехода должен был позволить коду, написанному до появления обобщений, остаться в рабочем состоянии и одновременно быть совместимым с обобщениями. Другими словами, коду, предшествующему обобщениям, нужна была возможность работы с обобщениями, а обобщенному коду — возможность работы с кодом, написанным до появления обобщений.

В плане обработки перехода к обобщениям Java разрешает использовать обобщенный класс без аргументов типов, что создает *низкоуровневый тип* для класса. Такой низкоуровневый тип совместим с унаследованным кодом, которому не известны обобщения. Главный недостаток применения низкоуровневого типа связан с утратой безопасности в отношении типов, обеспечиваемой обобщениями.

Ниже показан пример, демонстрирующий низкоуровневый тип в действии:

```
// Демонстрация низкоуровневого типа в действии.
class Gen<T> {
  T ob;  // объявить объект типа T

  // Передать конструктору ссылку на объект типа T.
  Gen(T o) {
    ob = o;
  }

  // Возвратить ob.
  T getOb() {
    return ob;
  }
}

// Использование низкоуровневого типа.
class RawDemo {
  public static void main(String[] args) {

    // Создать объект Gen для объектов Integer.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Создать объект Gen для объектов String.
    Gen<String> strOb = new Gen<String>("Тест обобщений");

    // Создать низкоуровневый объект Gen и предоставить ему значение Double.
    Gen raw = new Gen(Double.valueOf(98.6));

    // Приведение здесь обязательно, потому что тип неизвестен.
    double d = (Double) raw.getOb();
    System.out.println("значение: " + d);

    // Использование низкоуровневого типа может стать причиной
    // генерации исключений во время выполнения.
    // Вот несколько примеров.

    // Следующее приведение вызывает ошибку во время выполнения!
    // int i = (Integer) raw.getOb();  // ошибка во время выполнения
```

```
    // Это присваивание обходит механизм безопасности типов.
    strOb = raw;  // Нормально, но потенциально неправильно
    // String str = strOb.getOb();     // ошибка во время выполнения

    // Это присваивание обходит механизм безопасности типов.
    raw = iOb;    // Нормально, но потенциально неправильно
    // d = (Double) raw.getOb();       // ошибка во время выполнения
  }
}
```

В программе присутствуют несколько интересных вещей. Первым делом с помощью следующего объявления создается низкоуровневый тип обобщенного класса Gen:

```
Gen raw = new Gen(Double.valueOf(98.6));
```

Обратите внимание, что аргументы типов не указаны. По существу оператор создает объект Gen, тип T которого заменяется на Object.

Низкоуровневый тип не безопасен в отношении типов. Таким образом, переменной низкоуровневого типа можно присваивать ссылку на любой тип объекта Gen. Разрешено и обратное — переменной конкретного типа Gen может быть присвоена ссылка на низкоуровневый объект Gen. Однако обе операции потенциально небезопасны из-за того, что обходится механизм проверки типов обобщений.

Отсутствие безопасности в отношении типов иллюстрируется закомментированными строками в конце программы. Давайте разберем каждый случай. Для начала рассмотрим следующую ситуацию:

```
// int i = (Integer) raw.getOb();  // ошибка во время выполнения
```

В данном операторе получается значение ob внутри raw и приводится к Integer. Проблема в том, что raw содержит значение Double, а не целочисленное значение. Тем не менее, на этапе компиляции обнаружить это невозможно, поскольку тип raw неизвестен. В итоге оператор терпит неудачу во время выполнения.

В следующей кодовой последовательности переменной strOb (ссылке типа Gen<String>) присваивается ссылка на низкоуровневый объект Gen:

```
strOb = raw;  // Нормально, но потенциально неправильно
// String str = strOb.getOb();     // ошибка во время выполнения
```

Само по себе присваивание синтаксически корректно, но сомнительно. Так как переменная strOb имеет тип Gen<String>, предполагается, что она содержит строку. Однако после присваивания объект, на который ссылается strOb, содержит Double. Таким образом, когда во время выполнения предпринимается попытка присвоить содержимое strOb переменной str, возникает ошибка, потому что теперь strOb содержит Double. В результате при присваивании обобщенной ссылке низкоуровневой ссылки обходится механизм безопасности типов.

В показанной ниже кодовой последовательности предыдущий случай инвертируется:

```
raw = iOb;    // Нормально, но потенциально неправильно
// d = (Double) raw.getOb();      // ошибка во время выполнения
```

Здесь обобщенная ссылка присваивается низкоуровневой ссылке. Несмотря на правильность с точки зрения синтаксиса, могут возникнуть проблемы, как показано во второй строке. Теперь переменная `raw` ссылается на объект, который содержит объект `Integer`, но приведение предполагает, что она содержит объект `Double`. Проблема не может быть выявлена на этапе компиляции и взамен возникает ошибка во время выполнения.

Из-за потенциальной опасности, присущей низкоуровневым типам, компилятор `javac` отображает *непроверяемые предупреждения* при использовании низкоуровневого типа способом, который может поставить под угрозу безопасность к типам. Следующие строки в предыдущей программе приводят к выдаче компилятором непроверяемых предупреждений:

```
Gen raw = new Gen(Double.valueOf(98.6));
strOb = raw;  // Нормально, но потенциально неправильно
```

В первой строке вызывается конструктор `Gen` без аргумента типа, что инициирует предупреждение. Во второй строке низкоуровневая ссылка присваивается обобщенной переменной, что генерирует предупреждение.

Поначалу вы можете подумать, что показанная далее строка тоже должна приводит к выдаче непроверяемое предупреждения, но это не так:

```
raw = iOb;  // Нормально, но потенциально неправильно
```

Компилятор не генерирует никаких предупреждений, т.к. присваивание не приводит к *добавочной* потере безопасности типов, нежели та, что уже произошла при создании `raw`.

И последнее замечание: вы должны ограничить использование низкоуровневых типов ситуациями, когда вам нужно смешивать унаследованный код с более новым обобщенным кодом. Низкоуровневые типы являются просто переходным средством, а не тем, что следует применять в новом коде.

Иерархии обобщенных классов

Подобно необобщенным обобщенные классы могут быть частью иерархии классов. Таким образом, обобщенный класс может выступать в качестве суперкласса или быть подклассом. Ключевая разница между обобщенными и необобщенными иерархиями связана с тем, что в обобщенной иерархии любые аргументы типов, необходимые обобщенному суперклассу, должны передаваться вверх по иерархии всеми подклассами. Это похоже на способ передачи вверх по иерархии аргументов конструкторов.

Использование обобщенного суперкласса

Ниже показан простой пример иерархии, использующей обобщенный суперкласс:

```
// Простая иерархия обобщенных классов.
class Gen<T> {
  T ob;

  Gen(T o) {
    ob = o;
  }

  // Возвратить ob.
  T getOb() {
    return ob;
  }
}

// Подкласс Gen.
class Gen2<T> extends Gen<T> {
  Gen2(T o) {
    super(o);
  }
}
```

В этой иерархии Gen2 расширяет обобщенный класс Gen. Обратите внимание на объявление Gen2:

```
class Gen2<T> extends Gen<T> {
```

Параметр типа T указан в Gen2 и также передается Gen в конструкции extends, т.е. любой тип, переданный Gen2, тоже будет передаваться Gen. Например, следующее объявление передает Integer в качестве параметра типа в Gen:

```
Gen2<Integer> num = new Gen2<Integer>(100);
```

В итоге ob в порции Gen внутри Gen2 будет иметь тип Integer.

Обратите также внимание на то, что в Gen2 параметр типа T не применяется ни для каких целей кроме поддержки суперкласса Gen. Таким образом, даже если подкласс обобщенного суперкласса в противном случае не должен быть обобщенным, то он все равно должен указывать параметр типа или параметры типов, требуемые для его обобщенного суперкласса.

Конечно, при необходимости подкласс может добавлять собственные параметры типов. Например, вот вариант предыдущей иерархии, где Gen2 добавляет собственный параметр типа:

```
// Подкласс может добавлять собственные параметры типов.
class Gen<T> {
  T ob;  // объявить объект типа T

  // Передать конструктору ссылку на объект типа T.
  Gen(T o) {
    ob = o;
  }

  // Возвратить ob.
  T getOb() {
    return ob;
  }
}
```

```
// Подкласс Gen, определяющий второй параметр типа по имени V.
class Gen2<T, V> extends Gen<T> {
  V ob2;

  Gen2(T o, V o2) {
    super(o);
    ob2 = o2;
  }

  V getOb2() {
    return ob2;
  }
}

// Создать объект типа Gen2.
class HierDemo {
  public static void main(String[] args) {

    // Создать объект Gen2 для String и Integer.
    Gen2<String, Integer> x =
      new Gen2<String, Integer>("Значение: ", 99);

    System.out.print(x.getOb());
    System.out.println(x.getOb2());
  }
}
```

Взгляните на объявление данной версии Gen2:

```
class Gen2<T, V> extends Gen<T> {
```

Здесь T — тип, переданный в Gen, а V — тип, специфичный для Gen2. Параметр типа V используется для объявления объекта по имени ob2 и в качестве возвращаемого типа для метода getOb2(). В методе main() создается объект Gen2, в котором параметр типа T является String, а параметр типа V — Integer. Программа выводит вполне ожидаемый результат:

```
Значение: 99
```

Обобщенный подкласс

Необобщенный класс абсолютно законно может быть суперклассом обобщенного подкласса. Например, рассмотрим следующую программу:

```
// Необобщенный класс может быть суперклассом обобщенного подкласса.

// Необобщенный класс.
class NonGen {
  int num;

  NonGen(int i) {
    num = i;
  }

  int getnum() {
    return num;
  }
}
```

```
// Обобщенный подкласс.
class Gen<T> extends NonGen {
  T ob;  // объявить объект типа T

  // Передать конструктору ссылку на объект типа T.
  Gen(T o, int i) {
    super(i);
    ob = o;
  }

  // Возвратить ob.
  T getOb() {
    return ob;
  }
}

// Создать объект Gen.
class HierDemo2 {
  public static void main(String[] args) {

    // Создать объект Gen для String.
    Gen<String> w = new Gen<String>("Добро пожаловать", 47);

    System.out.print(w.getOb() + " ");
    System.out.println(w.getnum());
  }
}
```

Вот вывод, генерируемый программой:

```
Добро пожаловать 47
```

Обратите внимание в программе на то, как класс Gen наследуется от NonGen:

```
class Gen<T> extends NonGen {
```

Поскольку класс NonGen не является обобщенным, аргумент типа не указывается. Таким образом, хотя в Gen объявлен параметр типа T, он не нужен (и не может использоваться) в NonGen. В результате NonGen наследуется классом Gen в обычной манере без применения особых условий.

Сравнение типов в обобщенной иерархии во время выполнения

Вспомните операцию instanceof, предназначенную для получения информации о типе во время выполнения, которая была представлена в главе 13. Как объяснялось, операция instanceof определяет, является ли объект экземпляром класса. Она возвращает true, если объект имеет указанный тип или может быть приведен к указанному типу. Операцию instanceof можно применять к объектам обобщенных классов. В приведенном далее классе демонстрируются некоторые последствия совместимости типов обобщенной иерархии:

```
// Использование операции instanceof с иерархией обобщенных классов.
class Gen<T> {
  T ob;
```

```
  Gen(T o) {
    ob = o;
  }

  // Возвратить ob.
  T getOb() {
    return ob;
  }
}

// Подкласс Gen.
class Gen2<T> extends Gen<T> {
  Gen2(T o) {
    super(o);
  }
}

// Демонстрация последствий идентификации типов во время
// выполнения для иерархии обобщенных классов.
class HierDemo3 {
  public static void main(String[] args) {

    // Создать объект Gen для Integer.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Создать объект Gen2 для Integer.
    Gen2<Integer> iOb2 = new Gen2<Integer>(99);

    // Создать объект Gen2 для String.
    Gen2<String> strOb2 = new Gen2<String>("Тест обобщений");

    // Выяснить, является ли iOb2 какой-то формой Gen2.
    if(iOb2 instanceof Gen2<?>)
      System.out.println("iOb2 - экземпляр Gen2");

    // Выяснить, является ли iOb2 какой-то формой Gen.
    if(iOb2 instanceof Gen<?>)
      System.out.println("iOb2 - экземпляр Gen");

    System.out.println();

    // Выяснить, является ли strOb2 экземпляром Gen2.
    if(strOb2 instanceof Gen2<?>)
      System.out.println("strOb2 - экземпляр Gen2");

    // Выяснить, является ли strOb2 экземпляром Gen.
    if(strOb2 instanceof Gen<?>)
      System.out.println("strOb2 - экземпляр Gen");

    System.out.println();

    // Выяснить, является ли iOb экземпляром Gen2, что не так.
    if(iOb instanceof Gen2<?>)
      System.out.println("iOb - экземпляр Gen2");

    // Выяснить, является ли iOb экземпляром Gen, что так.
    if(iOb instanceof Gen<?>)
      System.out.println("iOb - экземпляр Gen");
  }
}
```

Вот вывод:

```
iOb2 - экземпляр Gen2
iOb2 - экземпляр Gen
strOb2 - экземпляр Gen2
strOb2 - экземпляр Gen
iOb - экземпляр Gen
```

В этой программе класс Gen2 определен как подкласс Gen, который является обобщенным по параметру типа T. В методе main() создаются три объекта: iOb — объект типа Gen<Integer>, iOb2 — экземпляр Gen2<Integer> и strOb2 — объект типа Gen2<String>. Затем в программе выполняются проверки с помощью instanceof для типа переменной iOb2:

```
// Выяснить, является ли iOb2 какой-то формой Gen2.
if(iOb2 instanceof Gen2<?>)
  System.out.println("iOb2 - экземпляр Gen2");
// Выяснить, является ли iOb2 какой-то формой Gen.
if(iOb2 instanceof Gen<?>)
  System.out.println("iOb2 - экземпляр Gen");
```

В выводе видно, что обе проверки успешны. В первой проверке тип переменной iOb2 сравнивается с Gen2<?>. Она проходит успешно, т.к. просто подтверждает, что iOb2 — объект некоторого типа объекта Gen2. Использование подстановочного знака позволяет операции instanceof выяснить, является ли iOb2 объектом любого типа Gen2. Далее тип переменной iOb2 проверяется на предмет принадлежности к Gen<?>, т.е. к типу суперкласса. Проверка тоже успешна, поскольку iOb2 — некоторая форма суперкласса Gen. Последующие несколько строк в main() повторяют ту же самую последовательность (с такими же результатами) для переменной strOb2. После этого в программе посредством показанных ниже строк проверяется переменная iOb, представляющая собой экземпляр Gen<Integer> (суперкласса):

```
// Выяснить, является ли iOb экземпляром Gen2, что не так.
if(iOb instanceof Gen2<?>)
  System.out.println("iOb - экземпляр Gen2");
// Выяснить, является ли iOb экземпляром Gen, что так.
if(iOb instanceof Gen<?>)
  System.out.println("iOb - экземпляр Gen");
```

Проверка в первом операторе if терпит неудачу, потому что тип iOb не является каким-то типом объекта Gen2. Проверка во втором операторе if проходит успешно, т.к. iOb — некоторый тип объекта Gen.

Приведение

Привести один экземпляр обобщенного класса к другому можно только в том случае, если во всем остальном они совместимы и их аргументы типов совпадают. Например, в контексте предыдущей программы следующее приведение будет допустимым, поскольку iOb2 включает экземпляр Gen<Integer>:

```
(Gen<Integer>) iOb2  // допустимо
```

Но показанное ниже приведение не считается допустимым, т.к. iOb2 не является экземпляром Gen<Long>:

```
(Gen<Long>) iOb2    // недопустимо
```

Переопределение методов в обобщенном классе

Подобно любому другому методу метод в обобщенном классе можно переопределять. Например, рассмотрим программу, в которой переопределен метод getOb():

```
// Переопределение обобщенного метода в обобщенном классе.
class Gen<T> {
  T ob;  // объявить объект типа T

  // Передать конструктору ссылку на объект типа T.
  Gen(T o) {
    ob = o;
  }

  // Возвратить ob.
  T getOb() {
    System.out.print("getOb() в Gen: " );
    return ob;
  }
}

// Подкласс Gen, в котором переопределяется getOb().
class Gen2<T> extends Gen<T> {

  Gen2(T o) {
    super(o);
  }

  // Переопределить getOb().
  T getOb() {
    System.out.print("getOb() в Gen2: ");
    return ob;
  }
}

// Демонстрация переопределения обобщенного метода.
class OverrideDemo {
  public static void main(String[] args) {

    // Создать объект Gen для Integer.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Создать объект Gen2 для Integers.
    Gen2<Integer> iOb2 = new Gen2<Integer>(99);

    // Создать объект Gen2 для Strings.
    Gen2<String> strOb2 = new Gen2<String> ("Тест обобщений");

    System.out.println(iOb.getOb());
    System.out.println(iOb2.getOb());
    System.out.println(strOb2.getOb());
  }
}
```

Вот вывод:

```
getOb() в Gen: 88
getOb() в Gen2: 99
getOb() в Gen2: Тест обобщений
```

Как подтверждает вывод, для объектов типа `Gen2` вызывается переопределенная версия `getOb()`, но для объектов типа `Gen` вызывается версия `getOb()` из суперкласса.

Выведение типов и обобщения

Начиная с версии JDK 7, появилась возможность сокращения синтаксиса, применяемого для создания экземпляра обобщенного типа. Для начала взгляните на следующий обобщенный класс:

```
class MyClass<T, V> {
  T ob1;
  V ob2;

  MyClass(T o1, V o2) {
    ob1 = o1;
    ob2 = o2;
  }
  // ...
}
```

До выхода JDK 7 для создания экземпляра `MyClass` нужно было использовать оператор вида:

```
MyClass<Integer, String> mcOb =
  new MyClass<Integer, String>(98, "Строка");
```

Здесь аргументы типов (`Integer` и `String`) указываются дважды: первый раз, когда объявляется `mcOb`, и второй — когда экземпляр `MyClass` создается через операцию `new`. Поскольку обобщения появились в JDK 5, такая форма необходима для всех версий Java, предшествующих JDK 7. Хотя с этой формой не связано ничего плохого, она чуть более многословна, чем должна быть. В конструкции `new` типы аргументов типов могут быть без труда выведены из типа `mcOb`, а потому нет никаких причин указывать их повторно. Чтобы принять соответствующие меры, в JDK 7 добавлен синтаксический элемент, позволяющий избежать второго указания.

Теперь предыдущее объявление можно переписать так:

```
MyClass<Integer, String> mcOb = new MyClass<>(98, "Строка");
```

Обратите внимание, что в части, где создается экземпляр, просто используются угловые скобки <>, которые представляют пустой список аргументов типов. Такую конструкцию называют *ромбовидной* операцией. Она сообщает компилятору о необходимости выведения аргументов типов для конструктора из выражения `new`. Основное преимущество этого синтаксиса выведения типов связано с тем, что он сокращает довольно длинные операторы объявлений.

Обобщим все упомянутое ранее. Когда применяется выведение типов, синтаксис объявления обобщенной ссылки и создания экземпляра имеет следующую общую форму:

```
class-name<type-arg-list> var-name = new class-name <>(cons-arg-list);
имя-класса<список-аргументов-типов> имя-переменной =
                        new имя-класса<>(список-аргументов-конструктора);
```

Здесь `class-name` — имя класса, `type-arg-list` — список аргументов типов, `var-name` — имя переменной, `cons-arg-list` — список аргументов конструктора. Список аргументов типов конструктора в `new` пуст.

Выведение типов можно также применять к передаче параметров. Например, если добавить в `MyClass` метод `isSame()`:

```
boolean isSame(MyClass<T, V> o) {
  if(ob1 == o.ob1 && ob2 == o.ob2) return true;
  else return false;
}
```

тогда показанный ниже вызов будет законным:

```
if(mcOb.isSame(new MyClass<>(1, "тест"))) System.out.println("Одинаковые");
```

В данном случае аргументы типов для аргумента, переданного методу `isSame()`, могут быть выведены из типа параметра.

В большинстве рассмотренных в книге примеров при объявлении экземпляров обобщенных классов по-прежнему будет использоваться полный синтаксис, что обеспечит работу примеров с любым компилятором Java, который поддерживает обобщения. Применение полного синтаксиса также дает очень четкое представление о том, что именно создается, а это крайне важно в коде примеров, предлагаемых в книге. Тем не менее, вы можете использовать синтаксис выведения типов в своем коде с целью упрощения имеющихся объявлений.

Выведение типов локальных переменных и обобщения

Как только что объяснялось, выведение типов уже поддерживается для обобщений за счет применения ромбовидной операции. Однако с обобщенным классом можно также использовать средство выведения типов локальных переменных, добавленное в JDK 10. Например, с учетом класса `MyClass`, определенного в предыдущем разделе, следующее объявление:

```
MyClass<Integer, String> mcOb =
  new MyClass<Integer, String>(98, "Строка");
```

можно переписать с применением средства выведения типов локальных переменных:

```
var mcOb = new MyClass<Integer, String>(98, "Строка");
```

В данном случае тип `mcOb` выводится как `MyClass<Integer, String>`, поскольку это тип его инициализатора. Вдобавок обратите внимание на то, что

использование var в результате дает более короткое объявление, нежели в противном случае. Обычно имена обобщенных типов могут оказываться довольно длинными и (иногда) сложными. Применение var — еще один способ существенно сократить такие объявления. По тем же причинам, которые только что объяснялись относительно ромбовидной операции, в оставшихся примерах, рассмотренных в книге, будет использоваться полный обобщенный синтаксис, но в вашем коде выведение типов локальных переменных может оказаться весьма полезным.

Стирание

Знать все детали о том, как компилятор Java преобразует исходный текст программы в объектный код, обычно не нужно. Тем не менее, в случае с обобщениями важно иметь некоторое общее представление о процессе, поскольку оно позволяет понять, по какой причине обобщенные средства работают именно так, как работают, и почему их поведение иногда немного удивляет. В связи с этим имеет смысл кратко обсудить реализацию обобщений в Java.

Важным ограничением, которое управляло способом добавления обобщений в Java, была необходимость поддержания совместимости с предыдущими версиями Java. Попросту говоря, обобщенный код обязан был быть совместимым с ранее существовавшим необобщенным кодом. Таким образом, любые изменения синтаксиса языка Java или машины JVM не должны были нарушать функционирование старого кода. Способ, которым в Java реализованы обобщения, удовлетворяющий упомянутому ограничению, предусматривает применение *стирания*.

Рассмотрим, каким образом работает стирание. При компиляции кода Java вся информация об обобщенных типах удаляется (стирается), что подразумевает замену параметров типов их ограничивающим типом, которым является Object, если не указано явное ограничение, и последующее применение надлежащих приведений (как определено аргументами типов) для обеспечения совместимости с типами, указанными в аргументах типов. Такая совместимость типов навязывается самим компилятором. Подход к обобщениям подобного рода означает, что параметры типов во время выполнения не существуют. Они попросту являются механизмом, относящимся к исходному коду.

Мостовые методы

Иногда компилятор вынужден добавлять в класс *мостовой метод* с целью обработки ситуаций, в которых стирание типа переопределенного метода в подклассе не производит такое же стирание, как соответствующий метод в суперклассе. В таком случае генерируется метод, который задействует стирание типов из суперкласса, и этот метод вызывает метод со стиранием типов, указанным в подклассе. Разумеется, мостовые методы встречаются только на уровне байт-кода; вы их не видите и не можете ими пользоваться.

Хотя мостовые методы не относятся к тому, чем вам обычно приходится заниматься, все же поучительно проанализировать ситуацию, когда они создаются. Взгляните на следующую программу:

```
// Ситуация, в которой создается мостовой метод.
class Gen<T> {
  T ob;  // объявить объект типа T

  // Передать конструктору ссылку на объект типа T.
  Gen(T o) {
    ob = o;
  }

  // Возвратить ob.
  T getOb() {
    return ob;
  }
}

// Подкласс Gen.
class Gen2 extends Gen<String> {

  Gen2(String o) {
    super(o);
  }

  // Переопределенная версия getOb(), специфичная для String.
  String getOb() {
    System.out.print("Вызван метод getOb(), специфичный для String: ");
    return ob;
  }
}

// Демонстрация ситуации, когда требуется мостовой метод.
class BridgeDemo {
  public static void main(String[] args) {

    // Создать объект Gen2 для String.
    Gen2 strOb2 = new Gen2("Тест обобщений");

    System.out.println(strOb2.getOb());
  }
}
```

Подкласс `Gen2` в программе расширяет класс `Gen`, но делает это с применением версии `Gen`, специфичной для типа `String`, как показано в его объявлении:

```
class Gen2 extends Gen<String> {
```

Кроме того, внутри класса `Gen2` метод `getOb()` переопределяется с указанием `String` в качестве возвращаемого типа:

```
// Переопределенная версия getOb(), специфичная для String.
String getOb() {
  System.out.print("Вызван метод getOb(), специфичный для String: ");
  return ob;
}
```

Все действия вполне приемлемы. Единственная трудность заключается в том, что из-за стирания типов ожидаемой формой `getOb()` будет:

```
Object getOb() {  // ...
```

Чтобы решить проблему, компилятор создает мостовой метод с показанной выше сигнатурой, который вызывает версию `getOb()` для типа `String`. Таким образом, просмотрев файл класса `Gen2` с помощью `javap`, вы заметите следующие методы:

```
class Gen2 extends Gen<java.lang.String> {
  Gen2(java.lang.String);
  java.lang.String getOb();
  java.lang.Object getOb();  // мостовой метод
}
```

Как видите, в файл класса `Gen2` был включен мостовой метод. (Комментарий добавлен автором, а не `javap`, и точный вывод может варьироваться в зависимости от используемой версии Java.)

Осталось сделать еще одно замечание по поводу рассмотренного примера. Обратите внимание на то, что единственная разница между двумя версиями метода `getOb()` связана с их возвращаемым типом. Обычно в таком случае происходит ошибка, но поскольку она не относится к исходному коду, проблема не возникает и корректно обрабатывается JVM.

Ошибки неоднозначности

Внедрение обобщений порождает еще одну разновидность ошибок, от которых вы должны защититься: *неоднозначность*. Ошибки неоднозначности происходят, когда стирание приводит к тому, что два на вид разных обобщенных объявления распознаются как один и тот же стертый тип, становясь причиной конфликта. Вот пример, касающийся перегрузки методов:

```
// Неоднозначность перегруженных методов, возникающая из-за стирания.
class MyGenClass<T, V> {
  T ob1;
  V ob2;

  // ...

  // Эти два перегруженных метода неоднозначны и не скомпилируются.
  void set(T o) {
    ob1 = o;
  }

  void set(V o) {
    ob2 = o;
  }
}
```

Обратите внимание, что в `MyGenClass` объявлены два обобщенных типа: `T` и `V`. Внутри `MyGenClass` предпринимается попытка перегрузки метода `set()` на основе параметров типов `T` и `V`, что выглядит разумным, поскольку `T` и `V` кажутся разными типами. Однако здесь имеются две проблемы неоднозначности.

Первая проблема заключается в том, что исходя из того, как написан класс MyGenClass, на самом деле вовсе не обязательно, чтобы T и V были разными типами. Скажем, абсолютно корректно (в принципе) сконструировать объект MyGenClass следующим образом:

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

В данном случае T и V будут заменены типом String. В итоге обе версии метода set() становятся идентичными, что, конечно же, является ошибкой.

Вторая и более фундаментальная проблема связана с тем, что стирание типов для set() сводит обе версии к такому виду:

```
void set(Object o) {  // ...
```

Соответственно перегрузка метода set(), предпринятая в MyGenClass, по своей сути неоднозначна.

Ошибки неоднозначности бывает сложно исправить. Например, если известно, что V всегда будет каким-то числовым типом, тогда вы можете попытаться исправить MyGenClass, переписав его объявление, как показано ниже:

```
class MyGenClass<T, V extends Number> {  // в основном нормально!
```

Внесенное изменение приводит к успешной компиляции MyGenClass, и можно даже создавать объекты:

```
MyGenClass<String, Number> x = new MyGenClass<String, Number>();
```

Такой код работает, потому что компилятор Java способен точно определить, какой метод вызывать. Тем не менее, неоднозначность снова возникает в следующей строке кода:

```
MyGenClass<Number, Number> x = new MyGenClass<Number, Number>();
```

Поскольку T и V имеют тип Number, какая версия set() в этом случае должна вызываться? Теперь неоднозначен вызов метода set().

Откровенно говоря, в предыдущем примере было бы гораздо лучше объявить два метода с отличающимися именами, нежели пытаться перегрузить set(). Часто решение проблемы неоднозначности предусматривает реструктуризацию кода, т.к. неоднозначность нередко означает наличие концептуальной ошибки в проекте.

Некоторые ограничения обобщений

Существует несколько ограничений, которые следует иметь в виду при работе с обобщениями. Они связаны с созданием объектов параметров типов, статических членов, исключений и массивов. Все ограничения исследуются ниже.

Невозможность создать экземпляры параметров типов

Создать экземпляр параметра типа невозможно. Например, рассмотрим следующий класс:

```
// Создать экземпляр T невозможно.
class Gen<T> {
  T ob;
  Gen() {
    ob = new T();  // Незаконно!!!
  }
}
```

Здесь предпринимается незаконная попытка создать экземпляр T. Понять причину должно быть легко: компилятору не известен тип объекта, который нужно создать. Параметр типа T — это просто заполнитель.

Ограничения, касающиеся статических членов

Статические члены не могут использовать параметр типа, объявленный в объемлющем классе. Например, оба статических члена этого класса недопустимы:

```
class Wrong<T> {
  // Ошибка, статические переменные не могут иметь тип T.
  static T ob;

  // Ошибка, статические методы не могут использовать T.
  static T getOb() {
    return ob;
  }
}
```

Хотя объявлять статические члены, в которых задействован параметр типа, объявленный в объемлющем классе, не разрешено, можно объявить статические обобщенные методы, которые определяют собственные параметры типов, как делалось ранее в главе.

Ограничения, касающиеся обобщенных массивов

Существуют два важных ограничения обобщений, которые применяются к массивам. Во-первых, нельзя создавать экземпляр массива, тип элементов которого является параметром типа. Во-вторых, невозможно создавать массив обобщенных ссылок для конкретного типа. Обе ситуации демонстрируются в показанной далее короткой программе:

```
// Обобщения и массивы.
class Gen<T extends Number> {
  T ob;

  T[] vals;  // нормально

  Gen(T o, T[] nums) {
    ob = o;
    // Этот оператор недопустим.
    // vals = new T[10];  // невозможно создать массив элементов типа T
    // Но следующий оператор законен.
    vals = nums;  // присваивать ссылку на существующий массив разрешено
  }
}
```

```
class GenArrays {
  public static void main(String[] args) {
    Integer[] n = { 1, 2, 3, 4, 5 };

    Gen<Integer> iOb = new Gen<Integer>(50, n);

    // Невозможно создать массив обобщенных ссылок для конкретного типа.
    // Gen<Integer>[] gens = new Gen<Integer>[10];  // Ошибка!

    // Все нормально.
    Gen<?>[] gens = new Gen<?>[10];                 // нормально
  }
}
```

Как показано в программе, объявлять ссылку на массив типа T разрешено:

```
T[] vals;           // нормально
```

Но создавать экземпляр массива элементов типа T нельзя, как демонстрируется в следующей закомментированной строке:

```
// vals = new T[10];// невозможно создать массив элементов типа T
```

Причина невозможности создать массив элементов типа T связана с тем, что компилятор не в состоянии выяснить фактический тип создаваемого массива.

Однако можно передать методу `Gen()` ссылку на совместимый по типу массив при создании объекта и присвоить эту ссылку переменной `vals`:

```
vals = nums;        // присваивать ссылку на существующий массив разрешено
```

Прием работает, поскольку переданный в `Gen` массив относится к известному типу, который будет совпадать с типом T во время создания объекта.

Обратите внимание, что внутри `main()` нельзя объявлять массив обобщенных ссылок для конкретного типа, т.е. следующая строка кода не скомпилируется:

```
// Gen<Integer>[] gens = new Gen<Integer>[10]; // Ошибка!
```

Но можно создать массив ссылок на обобщенный тип, если используется подстановочный знак:

```
Gen<?>[] gens = new Gen<?>[10];                 // нормально
```

Такой подход лучше применения массива низкоуровневых типов, потому что будет выполняться, по крайней мере, хоть какая-то проверка типов.

Ограничения, касающиеся обобщенных исключений

Обобщенный класс не может расширять тип `Throwable`, что означает невозможность создания обобщенных классов исключений.

ГЛАВА

15 Лямбда-выражения

В ходе непрерывного развития и эволюции языка Java, начиная с его первоначального выпуска 1.0, было добавлено много функциональных средств. Тем не менее, два из них выделяются тем, что они основательно трансформировали язык, коренным образом изменив способ написания кода. Первой фундаментальной модификацией стало добавление обобщений в JDK 5 (см. главу 14), а второй — внедрение *лямбда-выражений*, которые являются предметом настоящей главы.

Появившиеся в JDK 8 лямбда-выражения (и связанные с ними функции) значительно улучшили язык Java по двум главным причинам. Во-первых, они добавили новые элементы синтаксиса, которые увеличили выразительную мощь языка. В процессе они упростили способ реализации ряда общих конструкций. Во-вторых, добавление лямбда-выражений привело к включению в библиотеку Java API новых возможностей, среди которых возможность более легкой эксплуатации параллельной обработки в многоядерных средах, особенно в том, что касается обработки операций в стиле "for-each", и новый потоковый API, поддерживающий конвейерные операции с данными. Появление лямбда-выражений также послужило катализатором для других новых средств Java, включая стандартные методы (описанные в главе 9), которые позволяют определять поведение по умолчанию для методов интерфейса, и ссылки на методы (рассматриваемые здесь), позволяющие ссылаться на методы без их выполнения.

В конечном итоге следует отметить, что точно так же, как обобщения трансформировали Java несколько лет назад, лямбда-выражения продолжают изменять Java сегодня. Попросту говоря, лямбда-выражения окажут влияние практически на всех программистов на Java. Они действительно настолько важны.

Введение в лямбда-выражения

Ключевым аспектом для понимания реализации лямбда-выражений в Java являются две конструкции: собственно лямбда-выражение и функциональный интерфейс. Начнем с простого определения каждого из них.

Лямбда-выражение по существу представляет собой анонимный (т.е. безымянный) метод. Однако такой метод не выполняется сам по себе. Взамен он используется для реализации метода, определенного функциональным интерфейсом. Таким образом, лямбда-выражение приводит к форме анонимного класса. Лямбда-выражения также часто называют *замыканиями*.

Функциональный интерфейс — это интерфейс, который содержит один и только один абстрактный метод, обычно устанавливающий предполагаемое назначение интерфейса. Соответственно функциональный интерфейс, как правило, представляет одиночное действие. Например, стандартный интерфейс `Runnable` является функциональным интерфейсом, поскольку в нем определен только один метод: `run()`. Следовательно, `run()` определяет действие `Runnable`. Кроме того, функциональный интерфейс задает *целевой тип* лямбда-выражения. Важно понимать, что лямбда-выражение может применяться только в контексте, в котором указан его целевой тип. И еще один момент: функциональный интерфейс иногда называют *типом SAM*, где SAM означает Single Abstract Method — единственный абстрактный метод.

На заметку! В функциональном интерфейсе можно указывать любой открытый метод, определенный в `Object`, скажем, `equals()`, не влияя на его статус "функционального интерфейса". Открытые методы класса `Object` считаются неявными членами функционального интерфейса, потому что они реализуются экземпляром функционального интерфейса автоматически.

Теперь давайте займемся более подробным обсуждением лямбда-выражений и функциональных интерфейсов.

Основы лямбда-выражений

Лямбда-выражение вводит в язык Java новый синтаксический элемент и операцию. Новая операция иногда называется *лямбда-операцией* или *операцией стрелки* и обозначается с помощью `->`. Она делит лямбда-выражение на две части. В левой части указываются любые параметры, требующиеся в лямбда-выражении. (Если параметры не нужны, тогда используется пустой список параметров.) В правой части находится *тело лямбда-выражения*, которое определяет действия лямбда-выражения. Операцию `->` можно выразить словом "становится" либо "достается".

В Java определены два вида тела лямбда-выражения — с одиночным выражением и с блоком кода. Сначала мы рассмотрим лямбда-выражения, определяющие единственное выражение, а позже в главе займемся лямбда-выражениями, содержащими блочные тела.

Прежде чем продолжить, полезно взглянуть на несколько примеров лямбда-выражений. Давайте начнем с лямбда-выражения, пожалуй, самого простого вида, которое возможно записать. Результатом его вычисления будет константное значение:

```
() -> 123.45
```

Показанное выше лямбда-выражение не принимает параметров, из-за чего список параметров пуст, и возвращает константное значение 123.45. Таким образом, оно аналогично следующему методу:

```
double myMeth() { return 123.45; }
```

Разумеется, определяемый лямбда-выражением метод не имеет имени.

А вот более интересное лямбда-выражение:

```
() -> Math.random() * 100
```

Это лямбда-выражение получает псевдослучайное значение от Math.random(), умножает его на 100 и возвращает результат. Оно тоже не требует параметра.

Когда лямбда-выражению необходим параметр, он указывается в списке параметров слева от лямбда-операции, например:

```
(n) -> (n % 2)==0
```

Такое лямбда-выражение возвращает true, если значение параметра n оказывается четным. Хотя тип параметра (n в данном случае) допускается указывать явно, часто поступать так нет никакой необходимости, поскольку во многих случаях тип параметра может быть выведен. Подобно именованному методу в лямбда-выражении разрешено указывать любое нужное количество параметров.

Функциональные интерфейсы

Как упоминалось ранее, функциональный интерфейс представляет собой такой интерфейс, в котором определен только один абстрактный метод. Если вы уже какое-то время программировали на Java, то поначалу могли полагать, что все методы интерфейса неявно являются абстрактными. Хотя до выхода JDK 8 это было верно, впоследствии ситуация изменилась. Как объяснялось в главе 9, начиная с JDK 8, для объявляемого в интерфейсе метода можно указывать стандартную реализацию. Закрытые и статические методы интерфейса тоже обеспечивают реализацию. В результате теперь метод интерфейса будет абстрактным только в том случае, если для него не определена реализация. Поскольку нестандартные, нестатические, незакрытые методы интерфейса неявно абстрактны, нет нужды применять модификатор abstract (правда, при желании его можно указывать).

Ниже показан пример функционального интерфейса:

```
interface MyNumber {
  double getValue();
}
```

В данном случае метод getValue() является неявно абстрактным и единственным методом, определенным в MyNumber. Таким образом, MyNumber — функциональный интерфейс, функция которого определяется getValue().

Как упоминалось ранее, лямбда-выражение не выполняется само по себе. Оно скорее формирует реализацию абстрактного метода, определенного в

функциональном интерфейсе, который указывает его целевой тип. В результате лямбда-выражение может быть указано только в контексте, где определен целевой тип. Один из таких контекстов создается, когда лямбда-выражение присваивается ссылке на функциональный интерфейс. Другие контексты целевого типа включают помимо прочего инициализацию переменных, операторы `return` и аргументы метода.

Рассмотрим пример, где будет продемонстрировано, как можно использовать лямбда-выражение в контексте присваивания. Для начала объявляется ссылка на функциональный интерфейс `MyNumber`:

```
// Создать ссылку на интерфейс MyNumber.
MyNumber myNum;
```

Затем лямбда-выражение присваивается созданной ссылке на интерфейс:

```
// Использовать лямбда-выражение в контексте присваивания.
myNum = () -> 123.45;
```

Когда лямбда-выражение встречается в контексте целевого типа, автоматически создается экземпляр класса, который реализует функциональный интерфейс, а лямбда-выражение определяет поведение абстрактного метода, объявленного в функциональном интерфейсе. При вызове данного метода через цель лямбда-выражение выполняется. Таким образом, лямбда-выражение предоставляет способ трансформации кодового сегмента в объект.

В предыдущем примере лямбда-выражение становится реализацией метода `getValue()`. В результате следующий код отображает значение `123.45`:

```
// Вызвать метод getValue(), который реализован
// присвоенным ранее лямбда-выражением.
System.out.println(myNum.getValue());
```

Поскольку лямбда-выражение, присвоенное переменной `myNum`, возвращает значение `123.45`, это значение и будет получено при вызове `getValue()`.

Чтобы лямбда-выражение можно было применять в контексте целевого типа, типы абстрактного метода и лямбда-выражения должны быть совместимыми. Скажем, если в абстрактном методе заданы два параметра типа `int`, то лямбда-выражение должно указывать два параметра, тип которых — либо явный `int`, либо может быть неявно выведен контекстом как `int`. Обычно тип и количество параметров лямбда-выражения должны согласовываться с параметрами метода; возвращаемые типы должны быть совместимыми, а любые исключения, генерируемые лямбда-выражением, должны быть допустимыми для метода.

Примеры лямбда-выражений

С учетом предыдущего обсуждения давайте взглянем на несколько простых примеров, иллюстрирующих основные концепции лямбда-выражений. В первом примере собраны вместе все части, показанные в предыдущем разделе:

```
// Демонстрация использования простого лямбда-выражения.

// Функциональный интерфейс.
interface MyNumber {
  double getValue();
}

class LambdaDemo {
  public static void main(String[] args)
  {
    MyNumber myNum;  // объявить ссылку на интерфейс

    // Здесь лямбда-выражение представляет собой константное выражение.
    // Когда оно присваивается myNum, конструируется экземпляр класса,
    // где лямбда-выражение реализует метод getValue() из MyNumber.
    myNum = () -> 123.45;

    // Вызвать метод getValue(), предоставляемый ранее
    // присвоенным лямбда-выражением.
    System.out.println("Фиксированное значение: " + myNum.getValue());

    // Здесь используется более сложное лямбда-выражение.
    myNum = () -> Math.random() * 100;

    // В следующих операторах вызывается лямбда-выражение
    // из предыдущей строки кода.
    System.out.println("Случайное значение: " + myNum.getValue());
    System.out.println("Еще одно случайное значение: " + myNum.getValue());

    // Лямбда-выражение должно быть совместимым с методом,
    // определенным в функциональном интерфейсе.
    // Следовательно, показанный ниже код работать не будет:
    // myNum = () -> "123.03";  // Ошибка!
  }
}
```

Вот вывод:

```
Фиксированное значение: 123.45
Случайное значение: 88.90663650412304
Еще одно случайное значение: 53.00582701784129
```

Как уже упоминалось, лямбда-выражение должно быть совместимым с абстрактным методом, для реализации которого оно предназначено. По этой причине код в закомментированной строке в конце предыдущей программы недопустим, потому что значение типа `String` несовместимо с типом `double`, т.е. возвращаемым типом метода `getValue()`.

В следующем примере иллюстрируется использование параметра с лямбда-выражением:

```
// Демонстрация использования лямбда-выражения, принимающего параметр.

// Еще один функциональный интерфейс.
interface NumericTest {
  boolean test(int n);
}
class LambdaDemo2 {
  public static void main(String[] args)
  {
```

```
    // Лямбда-выражение, которое проверяет, четное ли число.
    NumericTest isEven = (n) -> (n % 2)==0;
    if(isEven.test(10)) System.out.println("10 -- четное число");
    if(!isEven.test(9)) System.out.println("9 -- нечетное число");
    // Лямбда-выражение, которое проверяет, является ли
    // число неотрицательным.
    NumericTest isNonNeg = (n) -> n >= 0;
    if(isNonNeg.test(1)) System.out.println("1 -- неотрицательное число");
    if(!isNonNeg.test(-1)) System.out.println("-1 -- отрицательное число");
  }
}
```

Ниже показан вывод, генерируемый программой:

```
10 -- четное число
9 -- нечетное число
1 -- неотрицательное число
-1 -- отрицательное число
```

В программе демонстрируется ключевой факт о лямбда-выражениях, требующий тщательного исследования. Обратите особое внимание на лямбда-выражение, которое выполняет проверку на четность:

```
(n) -> (n % 2)==0
```

Несложно заметить, что тип `n` не указан. Напротив, он выводится из контекста. В данном случае тип `n` выводится из типа параметра метода `test()`, как определено интерфейсом `NumericTest`, т.е. `int`. Кроме того, тип параметра в лямбда-выражении можно указывать явно. Например, вот тоже допустимый способ записи предыдущего лямбда-выражения:

```
(int n) -> (n % 2)==0
```

Здесь тип `n` явно указывается как `int`. Обычно задавать тип явным образом нет необходимости, но его можно указывать в ситуациях, когда это требуется. Начиная с JDK 11, разрешено также применять `var`, чтобы явно обозначить выведение типа локальной переменной для параметра лямбда-выражения.

В программе демонстрируется еще один важный момент, связанный с лямбда-выражениями: ссылка на функциональный интерфейс может использоваться для выполнения любого лямбда-выражения, которое с ним совместимо. Обратите внимание, что в программе определены два разных лямбда-выражения, совместимые с методом `test()` функционального интерфейса `NumericTest`. Первое лямбда-выражение, `isEven`, выясняет, является ли значение четным, а второе, `isNonNeg`, проверяет, является ли значение неотрицательным. В каждом случае проверяется значение параметра `n`. Поскольку каждое лямбда-выражение совместимо с методом `test()`, каждое из них может быть выполнено через ссылку `NumericTest`.

Прежде чем двигаться дальше, рассмотрим еще один момент. Когда лямбда-выражение имеет только один параметр, нет необходимости заключать имя параметра в круглые скобки, если оно указано в левой части лямбда-операции. Например, записать лямбда-выражение, применяемое в программе, можно и так:

```
n -> (n % 2)==0
```

В целях согласованности в книге все списки параметров лямбда-выражений будут заключаться в круглые скобки, даже те, которые содержат только один параметр. Конечно, вы можете предпочесть другой стиль.

В следующей программе демонстрируется использование лямбда-выражения, которое принимает два параметра. В этом случае оно проверяет, является ли одно число множителем другого.

```
// Демонстрация использования лямбда-выражения, принимающего два параметра.
interface NumericTest2 {
  boolean test(int n, int d);
}
class LambdaDemo3 {
  public static void main(String[] args)
  {
    // Это лямбда-выражение выясняет, является ли одно число
    // множителем другого.
    NumericTest2 isFactor = (n, d) -> (n % d) == 0;

    if(isFactor.test(10, 2))
      System.out.println("2 является множителем 10");
    if(!isFactor.test(10, 3))
      System.out.println("3 не является множителем 10");
  }
}
```

Ниже показан вывод:

```
2 является множителем 10
3 не является множителем 10
```

Функциональный интерфейс `NumericTest2` в программе определяет метод `test()`:

```
boolean test(int n, int d);
```

В данной версии для метода `test()` указаны два параметра. Таким образом, чтобы лямбда-выражение было совместимым с `test()`, оно тоже должно иметь два параметра. Обратите внимание на способ их указания:

```
(n, d) -> (n % d) == 0
```

Два параметра, `n` и `d`, указываются в списке параметров через запятую. Пример можно обобщить. Всякий раз, когда требуется более одного параметра, их необходимо указывать в списке внутри круглых скобок в левой части лямбда-операции, отделяя друг от друга запятыми.

Важно отметить один момент, касающийся множества параметров в лямбда-выражении: если нужно явно объявить тип параметра, тогда все параметры обязаны иметь объявленные типы. Скажем, следующий код допустим:

```
(int n, int d) -> (n % d) == 0
```

Но такой код — нет:

```
(int n, d) -> (n % d) == 0
```

Блочные лямбда-выражения

Тело в лямбда-выражениях, показанных в предшествующих примерах, состояло из единственного выражения. Такой вид тела лямбда-выражения называется *телом-выражением*, а лямбда-выражение с телом-выражением — *одиночным лямбда-выражением*. В теле-выражении код в правой части лямбда-операции должен содержать одно выражение. Хотя одиночные лямбда-выражения весьма полезны, иногда ситуация требует более одного выражения. Для обработки таких случаев в Java поддерживается второй вид лямбда-выражений, где в правой части лямбда-операции находится блок кода, который может содержать более одного оператора. Тело этого вида называется *блочным*. Лямбда-выражения с блочными телами иногда называются *блочными лямбда-выражениями*.

Блочное лямбда-выражение расширяет типы операций, которые могут быть обработаны в лямбда-выражении, поскольку позволяет телу лямбда-выражения содержать несколько операторов. Например, в блочном лямбда-выражении можно объявлять переменные, организовывать циклы, применять операторы `if` и `switch`, создавать вложенные блоки и т.д. Блочное лямбда-выражение создается легко. Нужно просто поместить тело в фигурные скобки подобно любому другому блоку операторов.

За исключением того, что блочные лямбда-выражения разрешают указывать несколько операторов, они используются почти так же, как только что рассмотренные одиночные лямбда-выражения. Тем не менее, есть одно ключевое отличие: вы обязаны явно применять оператор `return`, чтобы возвратить значение. Поступать так необходимо, потому что тело блочного лямбда-выражения не представляет одиночное выражение.

Вот пример, в котором используется блочное лямбда-выражение для вычисления и возврата факториала значения `int`:

```
// Блочное лямбда-выражение, которое вычисляет факториал значения int.
interface NumericFunc {
  int func(int n);
}

class BlockLambdaDemo {
  public static void main(String[] args)
  {
    // Это блочное лямбда-выражение вычисляет факториал значения int.
    NumericFunc factorial = (n) -> {
      int result = 1;

      for(int i=1; i <= n; i++)
        result = i * result;

      return result;
    };

    System.out.println("Факториал 3 равен " + factorial.func(3));
    System.out.println("Факториал 5 равен " + factorial.func(5));
  }
}
```

Вот вывод:

```
Факториал 3 равен 6
Факториал 5 равен 120
```

Обратите внимание в программе, что внутри блочного лямбда-выражения объявляется переменная по имени `result`, организуется цикл `for` и применяется оператор `return`. Они разрешены в теле блочного лямбда-выражения. По существу тело блочного лямбда-выражения похоже на тело метода. И еще один момент: когда в лямбда-выражении встречается оператор `return`, он просто приводит к возврату из лямбда-выражения, но не к возврату из объемлющего метода.

Ниже в программе предлагается другой пример блочного лямбда-выражения, которое изменяет порядок следования символов в строке на противоположный:

```
// Блочное лямбда-выражение, которое изменяет порядок
// следования символов в строке на противоположный.
interface StringFunc {
  String func(String n);
}

class BlockLambdaDemo2 {
  public static void main(String[] args)
  {
    // Это блочное лямбда-выражение изменяет порядок
    // следования символов в строке на противоположный.
    StringFunc reverse = (str) -> {
      String result = "";
      int i;

      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);

      return result;
    };

    System.out.println("Строка Lambda с противоположным порядком
следования символов: " +
                       reverse.func("Lambda"));
    System.out.println("Строка Expression с противоположным порядком
следования символов: " +
                       reverse.func("Expression"));
  }
}
```

Программа генерирует следующий вывод:

```
Строка Lambda с противоположным порядком следования символов: adbmaL
Строка Expression с противоположным порядком следования символов: noisserpxE
```

В этом примере в функциональном интерфейсе `StringFunc` объявлен метод `func()`, который принимает параметр типа `String` и возвращает тип `String`. Таким образом, в лямбда-выражении `reverse` тип `str` выводится как `String`. Обратите внимание, что метод `charAt()` вызывается на `str`, что допустимо из-за выведения типа `str` в `String`.

Обобщенные функциональные интерфейсы

Само лямбда-выражение не может указывать параметры типа. Таким образом, лямбда-выражение не может быть обобщенным. (Разумеется, из-за выведения типов все лямбда-выражения обладают некоторыми "обобщенными" качествами.) Однако функциональный интерфейс, ассоциированный с лямбда-выражением, может быть обобщенным. В таком случае целевой тип лямбда-выражения частично определяется аргументом или аргументами типов, указанными при объявлении ссылки на функциональный интерфейс.

Давайте попытаемся понять ценность обобщенных функциональных интерфейсов. В двух примерах из предыдущего раздела использовались два разных функциональных интерфейса: один назывался `NumericFunc`, а другой — `StringFunc`. Тем не менее, оба интерфейса определяли метод `func()`, который принимал один параметр и возвращал результат. В первом случае типом параметра и возвращаемого значения был `int`, а во втором случае — `String`. Таким образом, единственное отличие между двумя методами заключалось в типе требуемых данных. Вместо двух функциональных интерфейсов, методы которых различаются только типами данных, можно объявить один обобщенный интерфейс и применять его в обоих обстоятельствах. Подход демонстрируется в следующей программе:

```
// Использование обобщенного функционального интерфейса с лямбда-выражениями.

// Обобщенный функциональный интерфейс.
interface SomeFunc<T> {
  T func(T t);
}
class GenericFunctionalInterfaceDemo {
  public static void main(String[] args)
  {
    // Использовать версию String интерфейса SomeFunc.
    SomeFunc<String> reverse = (str) -> {
      String result = "";
      int i;
      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);
      return result;
    };
    System.out.println("Строка Lambda с противоположным порядком
следования символов: " +
                       reverse.func("Lambda"));
    System.out.println("Строка Expression с противоположным порядком
следования символов: " +
                       reverse.func("Expression"));
    // Теперь использовать версию Integer интерфейса SomeFunc.
    SomeFunc<Integer> factorial = (n) -> {
      int result = 1;
      for(int i=1; i <= n; i++)
        result = i * result;
      return result;
    };
```

```
    System.out.println("Факториал 3 равен " + factorial.func(3));
    System.out.println("Факториал 5 равен " + factorial.func(5));
  }
}
```

Вот вывод:

```
Строка Lambda с противоположным порядком следования символов: adbmaL
Строка Expression с противоположным порядком следования символов: noisserpxE
Факториал 3 равен 6
Факториал 5 равен 120
```

Обобщенный функциональный интерфейс `SomeFunc` объявлен в программе, как показано ниже:

```
interface SomeFunc<T> {
  T func(T t);
}
```

Здесь `T` указывает возвращаемый тип и тип параметра `func()`. Это означает, что он совместим с любым лямбда-выражением, которое принимает один параметр и возвращает значение того же самого типа.

Интерфейс `SomeFunc` используется для предоставления ссылки на два разных типа лямбда-выражений — `String` и `Integer`. Соответственно один и тот же функциональный интерфейс может применяться для ссылки на лямбда-выражения `reverse` и `factorial`. Отличается лишь аргумент типа, передаваемый в `SomeFunc`.

Передача лямбда-выражений в качестве аргументов

Как объяснялось ранее, лямбда-выражение можно использовать в любом контексте, который предоставляет целевой тип. Один из них касается передачи лямбда-выражения в виде аргумента. На самом деле передача лямбда-выражения в качестве аргумента является распространенным приемом применения лямбда-выражений. Более того, это очень эффективное использование, потому что оно дает возможность передавать исполняемый код в аргументе метода, значительно увеличивая выразительную мощь Java.

Чтобы лямбда-выражение можно было передавать как аргумент, тип параметра, получающего аргумент в форме лямбда-выражения, должен относиться к типу функционального интерфейса, который совместим с лямбда-выражением. Хотя применение лямбда-выражения в качестве аргумента сложностью не отличается, все-таки полезно увидеть его в действии. Процесс демонстрируется в следующей программе:

```
// Использование лямбда-выражений в качестве аргумента метода.
interface StringFunc {
  String func(String n);
}

class LambdasAsArgumentsDemo {
```

```
    // Типом первого параметра этого метода является функциональный интерфейс.
    // Таким образом, ему можно передавать ссылку на любой экземпляр реализации
    // данного интерфейса, в том числе экземпляр, созданный лямбда-выражением.
    // Во втором параметре указывается строка, с которой нужно работать.
    static String stringOp(StringFunc sf, String s) {
      return sf.func(s);
    }

    public static void main(String[] args)
    {
      String inStr = "Lambdas add power to Java";
      String outStr;

      System.out.println("Исходная строка: " + inStr);

      // Простое одиночное лямбда-выражение, которое переводит
      // в верхний регистр строку, переданную методу stringOp().
      outStr = stringOp((str) -> str.toUpperCase(), inStr);
      System.out.println("Строка в верхнем регистре: " + outStr);

      // Передать блочное лямбда-выражение, которое удаляет пробелы.
      outStr = stringOp((str) -> {
        String result = "";
        int i;

        for(i = 0; i < str.length(); i++)
          if(str.charAt(i) != ' ')
            result += str.charAt(i);

        return result;
      }, inStr);

      System.out.println("Строка после удаления пробелов: " + outStr);

      // Конечно, можно также передавать экземпляр StringFunc, заблаговременно
      // созданный лямбда-выражением. Например, после выполнения следующего
      // объявления reverse будет ссылаться на экземпляр StringFunc.
      StringFunc reverse = (str) -> {
        String result = "";
        int i;

        for(i = str.length()-1; i >= 0; i--)
          result += str.charAt(i);

        return result;
      };

      // Теперь reverse можно передать в первом параметре методу stringOp(),
      // поскольку этот параметр является ссылкой на объект StringFunc.
      System.out.println("Строка с противоположным порядком следования
символов: " + stringOp(reverse, inStr));
    }
  }
```

Ниже показан вывод:

```
Исходная строка: Lambdas add power to Java
Строка в верхнем регистре: LAMBDAS ADD POWER TO JAVA
Строка после удаления пробелов: LambdasaddpowertoJava
Строка с противоположным порядком следования символов: avaJ ot rewop dda
sadbmaL
```

Первым делом обратите внимание в программе на метод `stringOp()`, который принимает два параметра. Первый объявлен с типом `StringFunc`, который является функциональным интерфейсом. Таким образом, данный параметр может получать ссылку на любой экземпляр `StringFunc`, в том числе созданный лямбда-выражением. Второй параметр `stringOp()` имеет тип `String` и представляет собой строку, в отношении которой выполняется операция.

Теперь взгляните на первый вызов `stringOp()`:

```
outStr = stringOp((str) -> str.toUpperCase(), inStr);
```

Методу `stringOp()` в качестве аргумента передается простое одиночное лямбда-выражение, в результате чего создается экземпляр реализации функционального интерфейса `StringFunc`, ссылка на который передается в первом параметре `stringOp()`. Таким образом, методу передается лямбда-код, встроенный в экземпляр класса. Контекст целевого типа определяется типом параметра. Поскольку лямбда-выражение совместимо с этим типом, вызов будет допустимым. Встраивание простых лямбда-выражений вроде только что показанного в вызов метода часто оказывается удобным приемом, особенно когда лямбда-выражение предназначено для одноразового использования.

Далее методу `stringOp()` передается блочное лямбда-выражение, удаляющее пробелы из строки:

```
outStr = stringOp((str) -> {
  String result = "";
  int i;

  for(i = 0; i < str.length(); i++)
    if(str.charAt(i) != ' ')
      result += str.charAt(i);

  return result;
}, inStr);
```

Несмотря на применение блочного лямбда-выражения, процесс его передачи ничем не отличается от только что описанного процесса для одиночного лямбда-выражения. Однако в данном случае некоторые программисты сочтут синтаксис несколько неудобным.

Когда блочное лямбда-выражение кажется слишком длинным для встраивания в вызов метода, его легко присвоить переменной типа функционального интерфейса, как делалось в предшествующих примерах. Затем методу можно просто передать эту ссылку. Такой прием демонстрировался в конце программы, где определялось блочное лямбда-выражение, изменяющее порядок следования символов в строке на противоположный, и присваивалось переменной `reverse`, которая является ссылкой на экземпляр реализации `StringFunc`. Таким образом, `reverse` можно использовать как аргумент для первого параметра `stringOp()`. Затем в программе был вызван метод `stringOp()` с передачей ему переменной `reverse` и строки, с которой необходимо работать. Так как экземпляр, полученный при вычислении каждого

лямбда-выражения, представляет собой реализацию StringFunc, каждый из них разрешено применять в качестве первого аргумента для stringOp().

И последнее замечание: в дополнение к инициализации переменных, присваиванию и передаче аргументов контексты целевого типа создают приведения, операция ?, инициализаторы массивов, операторы return, а также сами лямбда-выражения.

Лямбда-выражения и исключения

Лямбда-выражение может генерировать исключение. Тем не менее, если инициируется проверяемое исключение, то оно должно быть совместимым с исключением или исключениями, которые перечислены в конструкции throws абстрактного метода в функциональном интерфейсе. Рассмотрим пример вычисления среднего значения для массива элементов типа double, иллюстрирующий данный факт. В случае передачи массива нулевой длины генерируется специальное исключение EmptyArrayException. Как показано в примере, исключение EmptyArrayException присутствует в конструкции throws метода func(), объявленного внутри функционального интерфейса DoubleNumericArrayFunc.

```
// Генерация исключения в лямбда-выражении.
interface DoubleNumericArrayFunc {
  double func(double[] n) throws EmptyArrayException;
}

class EmptyArrayException extends Exception {
  EmptyArrayException() {
    super("Массив пуст");
  }
}

class LambdaExceptionDemo {
  public static void main(String[] args) throws EmptyArrayException
  {
    double[] values = { 1.0, 2.0, 3.0, 4.0 };

    // Это блочное лямбда-выражение вычисляет среднее
    // значение для массива элементов типа double.
    DoubleNumericArrayFunc average = (n) ->  {
      double sum = 0;

      if(n.length == 0)
        throw new EmptyArrayException();

      for(int i=0; i < n.length; i++)
        sum += n[i];

      return sum / n.length;
    };
    System.out.println("Среднее значение равно " + average.func(values));
    // Следующий код приводит к генерации исключения.
    System.out.println("Среднее значение равно " + average.func(new double[0]));
  }
}
```

Первый вызов `Average.func()` возвращает значение `2.5`. Второй вызов, в котором передается массив нулевой длины, становится причиной генерации исключения `EmptyArrayException`. Не забывайте о необходимости включения конструкции `throws` в `func()`. Без нее программа не скомпилируется, потому что лямбда-выражение больше не будет совместимым с `func()`.

В примере демонстрируется еще один важный момент, касающийся лямбда-выражений. Обратите внимание, что параметр, указанный в методе `func()` функционального интерфейса `DoubleNumericArrayFunc`, объявлен как массив. Однако параметром лямбда-выражения является просто `n`, а не `n[]`. Помните о том, что тип параметра лямбда-выражения будет выводиться из целевого контекста. В этом случае целевой контекст — `double[]` и потому типом `n` окажется `double[]`. Указывать тип параметра в виде `n[]` не нужно, да и недопустимо. Было бы законно явно объявить параметр как `double[] n`, но здесь это ничего не даст.

Лямбда-выражения и захват переменных

Переменные, определенные в объемлющей области действия лямбда-выражения, доступны внутри лямбда-выражения. Скажем, лямбда-выражение может задействовать переменную экземпляра или статическую переменную, определенную в объемлющем классе. Лямбда-выражение также имеет доступ к ссылке `this` (явно и неявно), которая ссылается на вызывающий экземпляр класса, включающего лямбда-выражение. Таким образом, лямбда-выражение может получать или устанавливать значение переменной экземпляра или статической переменной и вызывать метод, определенный в объемлющем классе.

Тем не менее, когда в лямбда-выражении используется локальная переменная из его объемлющей области видимости, то возникает особая ситуация, называемая *захватом переменной*. В таком случае лямбда-выражение может работать только с локальными переменными, которые являются *фактически финальными*. Фактически финальная переменная представляет собой переменную, значение которой не меняется после ее первого присваивания. Явно объявлять такую переменную как `final` нет никакой необходимости, хотя поступать так не будет ошибкой. (Параметр `this` объемлющей области видимости автоматически будет фактически финальным, а лямбда-выражения не имеют собственной ссылки `this`.)

Важно понимать, что локальная переменная из объемлющей области не может быть модифицирована лямбда-выражением, поскольку в таком случае исчез бы ее статус фактически финальной, из-за чего она стала бы незаконной для захвата.

В следующей программе иллюстрируется отличие между фактически финальными и изменяемыми локальными переменными:

```
// Пример захвата локальной переменной из объемлющей области видимости.
interface MyFunc {
  int func(int n);
}
```

```
class VarCapture {
  public static void main(String[] args)
  {
    // Локальная переменная, которая может быть захвачена.
    int num = 10;

    MyFunc myLambda = (n) -> {
      // Использовать num подобным образом разрешено.
      // Переменная num не модифицируется.
      int v = num + n;

      // Однако следующая строка кода недопустима из-за того,
      // что она пытается модифицировать значение num.
      // num++;

      return v;
    };

    // Следующая строка кода тоже вызовет ошибку, потому что
    // она устранит статус переменной num как фактически финальной.
    // num = 9;
  }
}
```

В комментариях указано, что переменная num является фактически финальной и потому может применяться внутри myLambda. Однако если бы значение num изменилось внутри лямбда-выражения или за его пределами, то переменная num утратит свой статус фактически финальной, что вызовет ошибку и программа не скомпилируется.

Важно подчеркнуть, что лямбда-выражение может использовать и модифицировать переменную экземпляра из вызывающего его класса. Оно просто не может работать с локальной переменной из своей объемлющей области видимости, если только эта переменная не является фактически финальной.

Ссылки на методы

С лямбда-выражениями связана одна важная возможность, которая называется *ссылкой на метод*. Ссылка на метод предлагает способ обращения к методу, не инициируя его выполнение. Она имеет отношение к лямбда-выражениям, поскольку тоже требует контекста целевого типа, состоящего из совместимого функционального интерфейса. При вычислении ссылки на метод также создается экземпляр функционального интерфейса.

Существуют различные виды ссылок на методы. Мы начнем со ссылок на статические методы.

Ссылки на статические методы

Для ссылки на статический метод применяется следующий общий синтаксис:

имя-класса::*имя-метода*

Обратите внимание, что имя класса отделяется от имени метода двойным двоеточием. Разделитель :: был добавлен к языку в JDK 8 специально для этой цели. Ссылка на метод может использоваться везде, где она совместима со своим целевым типом.

В показанной далее программе демонстрируется применение ссылки на статический метод:

```
// Демонстрация использования ссылки на статический метод.

// Функциональный интерфейс для операций над строками.
interface StringFunc {
  String func(String n);
}

// В этом классе определен статический метод по имени strReverse().
class MyStringOps {
  // Статический метод, который изменяет порядок следования
  // символов на противоположный.
  static String strReverse(String str) {
    String result = "";
    int i;

    for(i = str.length()-1; i >= 0; i--)
      result += str.charAt(i);

    return result;
  }
}

class MethodRefDemo {

  // Первый параметр этого метода имеет тип функционального интерфейса.
  // Таким образом, ему можно передать любой экземпляр реализации
  // интерфейса StringFunc, включая ссылку на метод.
  static String stringOp(StringFunc sf, String s) {
    return sf.func(s);
  }

  public static void main(String[] args)
  {
    String inStr = "Lambdas add power to Java";
    String outStr;

    // Передать в stringOp() ссылку на статический метод strReverse().
    outStr = stringOp(MyStringOps::strReverse, inStr);

    System.out.println("Исходная строка: " + inStr);
    System.out.println("Строка с противоположным порядком следования
символов: " + outStr);
  }
}
```

Вот вывод:

```
Исходная строка: Lambdas add power to Java
Строка с противоположным порядком следования символов: avaJ ot rewop dda sadbmaL
```

Обратите особое внимание в программе на следующую строку:

```
outStr = stringOp(MyStringOps::strReverse, inStr);
```

Здесь методу stringOp() в качестве первого аргумента передается ссылка на статический метод strReverse(), объявленный внутри MyStringOps. Код работает по причине совместимости strReverse() с функциональным интерфейсом StringFunc. Таким образом, результатом вычисления выражения MyStringOps::strReverse будет ссылка на объект, в котором метод strReverse() предоставляет реализацию func() в StringFunc.

Ссылки на методы экземпляра

Чтобы передать ссылку на метод экземпляра конкретного объекта, используйте приведенный ниже базовый синтаксис:

объектная-ссылка::*имя-метода*

Как видите, синтаксис ссылки на метод экземпляра подобен синтаксису, применяемому для ссылки на статический метод, но вместо имени класса используется объектная ссылка. Предыдущую программу можно переписать с целью применения ссылки на метод экземпляра:

```
// Демонстрация использования ссылки на метод экземпляра.

// Функциональный интерфейс для операций над строками.
interface StringFunc {
  String func(String n);
}

// Теперь в этом классе определен метод экземпляра по имени strReverse().
class MyStringOps {
  String strReverse(String str) {
    String result = "";
    int i;

    for(i = str.length()-1; i >= 0; i--)
      result += str.charAt(i);

    return result;
  }
}

class MethodRefDemo2 {

  // Первый параметр этого метода имеет тип функционального интерфейса.
  // Таким образом, ему можно передавать любой экземпляр реализации
  // интерфейса StringFunc, включая ссылку на метод.
  static String stringOp(StringFunc sf, String s) {
    return sf.func(s);
  }

  public static void main(String[] args)
  {
    String inStr = "Lambdas add power to Java";
    String outStr;

    // Создать объект MyStringOps.
    MyStringOps strOps = new MyStringOps();

    // Передать в stringOp() ссылку на метод экземпляра strReverse().
    outStr = stringOp(strOps::strReverse, inStr);
```

```
      System.out.println("Исходная строка: " + inStr);
      System.out.println("Строка с противоположным порядком следования символов: "
+ outStr);
    }
  }
```

Программа генерирует тот же вывод, что и ее предыдущая версия.

Обратите внимание в программе, что `strReverse()` теперь является методом экземпляра `MyStringOps`. Внутри `main()` создается экземпляр `MyStringOps` по имени `strOps`, который используется для создания ссылки на метод `strReverse()` при обращении к `stringOp`:

```
outStr = stringOp(strOps::strReverse, inStr);
```

В этом примере `strReverse()` вызывается на объекте `strOps`.

Возможна также ситуация, когда желательно указать метод экземпляра, который можно применять с любым объектом заданного класса, а не только с указанным объектом. В таком случае ссылку на метод необходимо создать, как показано ниже:

```
имя-класса::имя-метода-экземпляра
```

Здесь вместо конкретного объекта используется имя класса, даже когда указан метод экземпляра. В такой форме первый параметр функционального интерфейса соответствует вызываемому объекту, а второй — параметру, заданному методом. Рассмотрим пример, где определяется метод `counter()`, подсчитывающий число объектов в массиве, которые удовлетворяют условию, определенному методом `func()` функционального интерфейса `MyFunc`. В этом случае он подсчитывает количество экземпляров класса `HighTemp`.

```
// Использование ссылки на метод экземпляра с разными объектами.

// Функциональный интерфейс с методом, который получает два
// ссылочных аргумента и возвращает булевский результат.
interface MyFunc<T> {
  boolean func(T v1, T v2);
}

// Класс, предназначенный для хранения самой высокой температуры за сутки.
class HighTemp {
  private int hTemp;

  HighTemp(int ht) { hTemp = ht; }

  // Возвратить true, если вызывающий объект HighTemp
  // содержит такую же температуру, как у ht2.
  boolean sameTemp(HighTemp ht2) {
    return hTemp == ht2.hTemp;
  }

  // Возвратить true, если вызывающий объект HighTemp
  // содержит температуру, которая ниже, чем у ht2.
  boolean lessThanTemp(HighTemp ht2) {
    return hTemp < ht2.hTemp;
  }
}
```

```
class InstanceMethWithObjectRefDemo {
  // Метод, возвращающий количество вхождений объекта, для которого
  // выполняются некоторые критерии, указанные параметром MyFunc.
  static <T> int counter(T[] vals, MyFunc<T> f, T v) {
    int count = 0;

    for(int i=0; i < vals.length; i++)
      if(f.func(vals[i], v))
        count++;

    return count;
  }

  public static void main(String[] args)
  {
    int count;

    // Создать массив объектов HighTemp.
    HighTemp[] weekDayHighs = { new HighTemp(89), new HighTemp(82),
                                new HighTemp(90), new HighTemp(89),
                                new HighTemp(89), new HighTemp(91),
                                new HighTemp(84), new HighTemp(83) };

    // Использовать counter() с массивами элементов типа HighTemp.
    // Обратите внимание, что во втором аргументе передается ссылка
    // на метод экземпляра sameTemp().
    count = counter(weekDayHighs, HighTemp::sameTemp, new HighTemp(89));
    System.out.println("Количество суток, когда самая высокая температура
была 89 градусов:" +
                       count);

    // Создать и использовать еще один массив элементов типа HighTemp.
    HighTemp[] weekDayHighs2 = { new HighTemp(32), new HighTemp(12),
                                 new HighTemp(24), new HighTemp(19),
                                 new HighTemp(18), new HighTemp(12),
                                 new HighTemp(-1), new HighTemp(13) };

    count = counter(weekDayHighs2, HighTemp::sameTemp, new HighTemp(12));
    System.out.println("Количество суток, когда самая высокая температура
была 12 градусов:" + count);

    // Использовать lessThanTemp() для нахождения суток, когда температура
    // была ниже указанного значения.
    count = counter(weekDayHighs, HighTemp::lessThanTemp, new HighTemp(89));
    System.out.println("Количество суток, когда самая высокая температура
была меньше 89 градусов:" + count);

    count = counter(weekDayHighs2, HighTemp::lessThanTemp, new HighTemp(19));
    System.out.println("Количество суток, когда самая высокая температура
была меньше 19 градусов:" + count);
  }
}
```

Вот вывод, генерируемый программой:

```
Количество суток, когда самая высокая температура была 89 градусов: 3
Количество суток, когда самая высокая температура была 12 градусов: 2
Количество суток, когда самая высокая температура была меньше 89 градусов: 3
Количество суток, когда самая высокая температура была меньше 19 градусов: 5
```

Обратите внимание в программе, что класс HighTemp имеет два метода экземпляра: sameTemp() и lessThanTemp(). Метод sameTemp() возвращает true, если два объекта HighTemp содержат ту же самую температуру. Метод lessThanTemp() возвращает true, если температура вызывающего объекта меньше температуры переданного объекта. Оба метода принимают параметр типа HighTemp и возвращают булевский результат. Следовательно, каждый метод совместим с функциональным интерфейсом MyFunc, поскольку тип вызывающего объекта может быть сопоставлен с первым параметром func(), а аргумент — со вторым параметром func(). Таким образом, когда следующее выражение:

```
HighTemp::sameTemp
```

передается методу counter(), создается экземпляр реализации функционального интерфейса MyFunc, в котором тип параметра первого параметра соответствует типу вызывающего объекта метода экземпляра, т.е. HighTemp. Типом второго параметра также будет HighTemp, потому что это тип параметра sameTemp(). То же самое утверждение справедливо и для метода lessThanTemp(). Еще один момент: с помощью super можно ссылаться на версию метода из суперкласса:

```
super::имя
```

Имя метода указывается в имя. Другая форма выглядит так:

```
имя-типа.super::имя
```

где имя-типа относится к объемлющему классу или суперинтерфейсу.

Ссылки на методы и обобщения

Ссылки на методы можно применять с обобщенными классами и/или обобщенными методами. Например, взгляните на следующую программу:

```
// Демонстрация использования ссылки на обобщенный метод,
// объявленный внутри необобщенного класса.

// Функциональный интерфейс, который работает с массивом
// и значением и возвращает результат int.
interface MyFunc<T> {
  int func(T[] vals, T v);
}
// В этом классе определен метод по имени countMatching(), который
// возвращает количество элементов в массиве, равных указанному значению.
// Обратите внимание, что метод countMatching() является обобщенным,
// но класс MyArrayOps - нет.
class MyArrayOps {
  static <T> int countMatching(T[] vals, T v) {
    int count = 0;

    for(int i=0; i < vals.length; i++)
      if(vals[i] == v)
        count++;
    return count;
  }
}
```

```
class GenericMethodRefDemo {

  // Первый параметр этого метода имеет тип функционального интерфейса MyFunc.
  // В остальных двух параметрах он принимает массив и значение, оба типа T.
  static <T> int myOp(MyFunc<T> f, T[] vals, T v) {
    return f.func(vals, v);
  }

  public static void main(String[] args)
  {
    Integer[] vals = { 1, 2, 3, 4, 2, 3, 4, 4, 5 };
    String[] strs = { "One", "Two", "Three", "Two" };
    int count;

    count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);
    System.out.println("Количество элементов 4, содержащихся в vals: "
      + count);

    count = myOp(MyArrayOps::<String>countMatching, strs, "Two");
    System.out.println("Количество элементов Two, содержащихся в strs: "
      + count);
  }
}
```

Ниже показан вывод:

```
Количество элементов 4, содержащихся в vals: 3
Количество элементов Two, содержащихся в strs: 2
```

В программе определен `MyArrayOps` — необобщенный класс, содержащий обобщенный метод по имени `countMatching()`, который возвращает количество элементов в массиве, равных заданному значению. Обратите внимание на способ указания аргумента обобщенного типа. Например, при первом вызове в `main()` ему передается аргумент типа `Integer`:

```
count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);
```

Вызов находится после `::`. Такой синтаксис можно универсализировать: когда обобщенный метод указывается через ссылку на метод, его аргумент типа идет после `::` и перед именем метода. Тем не менее, важно отметить, что явное указание аргумента типа в этой ситуации (и во многих других) не требуется, поскольку аргумент типа был бы выведен автоматически. В случаях, когда указан обобщенный класс, аргумент типа следует за именем класса и предшествует `::`.

Хотя в предыдущих примерах демонстрировался механизм использования ссылок на методы, их реальные преимущества не были отражены. Ссылки на методы могут оказаться весьма полезными в сочетании с инфраструктурой Collections Framework, которая описана в главе 20. Однако для полноты картины мы рассмотрим короткий, но эффективный пример, в котором ссылка на метод применяется для определения самого большого элемента в коллекции. (Если вы не знакомы с инфраструктурой Collections Framework, тогда возвратитесь к этому примеру после проработки материала главы 20.)

Один из способов нахождения самого большого элемента в коллекции предусматривает использование метода `max()`, определенного в классе

Collections. Вызываемой здесь версии метода max() необходимо передать ссылку на коллекцию и экземпляр объекта, реализующего интерфейс Comparator<T>, который устанавливает, каким образом сравниваются два объекта. В нем определен только один абстрактный метод compare(), который принимает два аргумента, имеющие типы сравниваемых объектов. Он должен возвращать значение больше нуля, если первый аргумент больше второго, ноль, если два аргумента равны, и значение меньше нуля, если первый объект меньше второго.

В прошлом для применения метода max() с объектами, определенными пользователем, нужно было получить экземпляр реализации интерфейса Comparator<T>, явно реализовав его классом, создав экземпляр этого класса и передав его max() в качестве компаратора. Начиная с версии JDK 8, можно просто передать в max() ссылку на метод сравнения, поскольку в таком случае компаратор реализуется автоматически. В следующем простом примере иллюстрируется процесс создания коллекции ArrayList объектов MyClass и последующего поиска в ней объекта, который содержит наибольшее значение (как определено методом сравнения).

```
// Использование ссылки на метод при поиске максимального значения в коллекции.
import java.util.*;

class MyClass {
  private int val;

  MyClass(int v) { val = v; }

  int getVal() { return val; }
}

class UseMethodRef {
  // Метод compareMC(), совместимый с методом compare(),
  // который определен в Comparator<T>.
  static int compareMC(MyClass a, MyClass b) {
    return a.getVal() - b.getVal();
  }
  public static void main(String[] args)
  {
    ArrayList<MyClass> al = new ArrayList<MyClass>();

    al.add(new MyClass(1));
    al.add(new MyClass(4));
    al.add(new MyClass(2));
    al.add(new MyClass(9));
    al.add(new MyClass(3));
    al.add(new MyClass(7));

    // Найти максимальное значение в al, используя метод compareMC().
    MyClass maxValObj = Collections.max(al, UseMethodRef::compareMC);

    System.out.println("Максимальное значение равно: " + maxValObj.getVal());
  }
}
```

Вот вывод:

```
Максимальное значение равно: 9
```

Обратите внимание в программе, что в самом классе MyClass никаких собственных методов сравнения не определено и не реализован интерфейс Comparator. Тем не менее, максимальное значение в списке элементов MyClass по-прежнему можно получить, вызывая метод max(), т.к. в классе UseMethodRef определен статический метод compareMC(), который совместим с методом compare(), определенным в Comparator. Следовательно, явно реализовывать и создавать экземпляр реализации Comparator не придется.

Ссылки на конструкторы

Подобно ссылкам на методы можно создавать ссылки на конструкторы. Ниже приведена общая форма синтаксиса, предназначенного для создания ссылки на конструктор:

имя-класса::new

Такую ссылку можно присваивать любой ссылке на функциональный интерфейс, в котором определен метод, совместимый с конструктором. Рассмотрим простой пример:

```
// Демонстрация использования ссылки на конструктор.

// MyFunc - функциональный интерфейс, метод которого
// возвращает ссылку на конструктор MyClass.
interface MyFunc {
  MyClass func(int n);
}

class MyClass {
  private int val;

  // Конструктор, принимающий аргумент.
  MyClass(int v) { val = v; }

  // Стандартный конструктор.
  MyClass() { val = 0; }
  // ...
  int getVal() { return val; };
}

class ConstructorRefDemo {
  public static void main(String[] args)
  {
    // Создать ссылку на конструктор MyClass.
    // Поскольку метод func() в MyFunc принимает аргумент, new ссылается
    // на параметризованный конструктор MyClass, а не на стандартный.
    MyFunc myClassCons = MyClass::new;

    // Создать экземпляр MyClass через эту ссылку на конструктор.
    MyClass mc = myClassCons.func(100);

    // Использовать только что созданный экземпляр MyClass.
    System.out.println("val в mc равно " + mc.getVal());
  }
}
```

Вот вывод:

```
val в mc равно 100
```

Как видите, метод func() класса MyFunc в программе возвращает ссылку типа MyClass и принимает параметр int. Кроме того, в MyClass определены два конструктора. В первом имеется параметр типа int, а второй является стандартным конструктором без параметров. Теперь взгляните на следующую строку:

```
MyFunc myClassCons = MyClass::new;
```

Выражение MyClass::new создает ссылку на конструктор MyClass. Поскольку в данном случае метод func() из MyFunc принимает параметр int, ссылка производится на конструктор MyClass(int v), т.к. он обеспечивает соответствие. Вдобавок обратите внимание, что ссылка на этот конструктор присваивается ссылке MyFunc по имени myClassCons. После выполнения оператора переменную myClassCons можно использовать для создания экземпляра MyClass:

```
MyClass mc = myClassCons.func(100);
```

По существу myClassCons становится еще одним способом вызова MyClass(int v).

Ссылки на конструкторы обобщенных классов создаются аналогичным образом. Единственное отличие связано с возможностью указания аргумента типа. Как и в случае применения обобщенного класса для создания ссылки на метод, аргумент типа указывается после имени класса. Далее предыдущий пример будет модифицирован, чтобы сделать MyFunc и MyClass обобщенными.

```
// Демонстрация использования ссылки на конструктор обобщенного класса.

// Теперь MyFunc - обобщенный функциональный интерфейс.
interface MyFunc<T> {
  MyClass<T> func(T n);
}
class MyClass<T> {
  private T val;

  // Конструктор, принимающий аргумент.
  MyClass(T v) { val = v; }

  // Стандартный конструктор.
  MyClass() { val = null;     }

  // ...

  T getVal() { return val; };
}
class ConstructorRefDemo2 {
  public static void main(String[] args)
  {
    // Создать ссылку на конструктор MyClass<T>.
    MyFunc<Integer> myClassCons = MyClass<Integer>::new;

    // Создать экземпляр MyClass<T> через эту ссылку на конструктор.
    MyClass<Integer> mc = myClassCons.func(100);
```

```
    // Использовать только что созданный экземпляр MyClass<T>.
    System.out.println("val в mc равно " + mc.getVal());
  }
}
```

Новая версия программы выдает тот же результат, что и предыдущая версия. Разница в том, что теперь `MyFunc` и `MyClass` определены как обобщенные. Таким образом, последовательность создания ссылки на конструктор может включать аргумент типа (хотя он нужен не всегда):

```
MyFunc<Integer> myClassCons = MyClass<Integer>::new;
```

Из-за того, что аргумент типа `Integer` уже был указан при создании `myClassCons`, его можно использовать для создания объекта `MyClass<Integer>`:

```
MyClass<Integer> mc = myClassCons.func(100);
```

Хотя в предшествующих примерах демонстрировался механизм работы со ссылкой на конструктор, реально использовать ее в подобной манере никто не будет, т.к. это не сулит никакой выгоды. Кроме того, наличие двух имен для одного и того же конструктора порождает безо всякого преувеличения запутанную ситуацию. Однако чтобы вы получили представление о более практичном применении, в следующей программе используется статический метод по имени `myClassFactory()`, который реализует фабрику для объектов `MyFunc` любого типа. Его можно задействовать при создании любого типа объекта, который имеет конструктор, совместимый с первым параметром `myClassFactory()`.

```
// Реализация простой фабрики классов с использованием ссылки на конструктор.
interface MyFunc<R, T> {
  R func(T n);
}
// Простой обобщенный класс.
class MyClass<T> {
  private T val;

  // Конструктор, принимающий аргумент.
  MyClass(T v) { val = v; }

  // Стандартный конструктор. В этой программе НЕ используется.
  MyClass() { val = null; }
  // ...
  T getVal() { return val; };
}

// Простой необобщенный класс.
class MyClass2 {
  String str;

  // Конструктор, принимающий аргумент.
  MyClass2(String s) { str = s; }

  // Стандартный конструктор. В этой программе НЕ используется.
  MyClass2() { str = ""; }
  // ...
  String getVal() { return str; };
}
```

```
class ConstructorRefDemo3 {

  // Фабричный метод для объектов класса. Класс обязан иметь
  // конструктор, который принимает один параметр типа T.
  // Тип создаваемого объекта указывается в R.
  static <R,T> R myClassFactory(MyFunc<R, T> cons, T v) {
    return cons.func(v);
  }

  public static void main(String[] args)
  {
    // Создать ссылку на конструктор MyClass.
    // В этом случае new ссылается на конструктор, принимающий аргумент.
    MyFunc<MyClass<Double>, Double> myClassCons = MyClass<Double>::new;

    // Создать экземпляр MyClass с применением фабричного метода.
    MyClass<Double> mc = myClassFactory(myClassCons, 100.1);

    // Использовать только что созданный экземпляр MyClass.
    System.out.println("val в mc равно " + mc.getVal());

    // Теперь создать другой класс с применением myClassFactory().
    MyFunc<MyClass2, String> myClassCons2 = MyClass2::new;

    // Создать экземпляр MyClass2, используя фабричный метод.
    MyClass2 mc2 = myClassFactory(myClassCons2, "Lambda");

    // Использовать только что созданный экземпляр MyClass2.
    System.out.println("str в mc2 равно " + mc2.getVal());
  }
}
```

Ниже показан вывод, генерируемый программой:

```
val в mc равно 100.1
str в mc2 равно Lambda
```

Как видите, статический метод `myClassFactory()` применяется для создания объектов типа `MyClass<Double>` и `MyClass2`. Хотя оба класса различаются, скажем, `MyClass` является обобщенным, а `MyClass2` — нет, с помощью `myClassFactory()` могут создаваться объекты обоих классов, т.к. они имеют конструкторы, совместимые с `func()` в `MyFunc`. Прием работает, потому что методу `myClassFactory()` передается конструктор строящегося объекта. Возможно, вам захочется немного поэкспериментировать с программой, испытав разные созданные ранее классы. Также попробуйте создать экземпляры различных типов объектов `MyClass`. Вы заметите, что метод `myClassFactory()` способен создавать любой объект, класс которого имеет конструктор, совместимый с `func()` в `MyFunc`. Хотя приведенный пример довольно прост, он раскрывает всю мощь, привносимую ссылками на конструкторы в язык Java.

Прежде чем двигаться дальше, важно упомянуть о существовании второй формы синтаксиса ссылок на конструкторы, которая предназначена для массивов. Чтобы создать ссылку на конструктор для массива, используйте такую конструкцию:

```
тип[]::new
```

Здесь посредством тип указывается тип создаваемого объекта. Например, при наличии определения MyClass из первого примера работы со ссылкой на конструктор (ConstructorRefDemo) и с учетом следующего интерфейса MyArrayCreator:

```
interface MyArrayCreator<T> {
  T func(int n);
}
```

вот как создать двухэлементный массив объектов MyClass и присвоить каждому начальное значение:

```
MyArrayCreator<MyClass[]> mcArrayCons = MyClass[]::new;
MyClass[] a = mcArrayCons.func(2);
a[0] = new MyClass(1);
a[1] = new MyClass(2);
```

Вызов func(2) приводит к созданию двухэлементного массива. Как правило, функциональный интерфейс обязан содержать метод с единственным параметром типа int, если он должен применяться для ссылки на конструктор массива.

Предопределенные функциональные интерфейсы

Вплоть до этого момента в примерах настоящей главы определялись собственные функциональные интерфейсы, что позволило четко проиллюстрировать фундаментальные концепции, лежащие в основе лямбда-выражений и функциональных интерфейсов. Тем не менее, во многих случаях определять собственный функциональный интерфейс не понадобится, поскольку в пакете java.util.function предлагается ряд предопределенных интерфейсов. Хотя они более подробно рассматриваются в части II, в табл. 15.1 описано несколько таких интерфейсов.

В показанной ниже программе демонстрируется работа интерфейса Function. В более раннем примере под названием BlockLambdaDemo вычислялся факториал с использованием блочных лямбда-выражений и созданием собственного функционального интерфейса по имени NumericFunc. Однако можно было бы задействовать встроенный интерфейс Function, как сделано в новой версии программы:

```
// Использование встроенного функционального интерфейса Function.

// Импортировать интерфейс Function.
import java.util.function.Function;

class UseFunctionInterfaceDemo {
  public static void main(String[] args)
  {

    // Это блочное лямбда-выражение вычисляет факториал значения int.
    // Теперь функциональным интерфейсом является Function.
```

```
    Function<Integer, Integer> factorial = (n) -> {
      int result = 1;
      for(int i=1; i <= n; i++)
        result = i * result;
      return result;
    };
    System.out.println("Факториал 3 равен " + factorial.apply(3));
    System.out.println("Факториал 5 равен " + factorial.apply(5));
  }
}
```

Программа выдает такой же вывод, как ее предыдущая версия.

Таблица 15.1. Избранные предопределенные функциональные интерфейсы

Интерфейс	Описание
`UnaryOperator<T>`	Применяет унарную операцию к объекту типа `T` и возвращает результат тоже типа `T`. Его метод называется `apply()`
`BinaryOperator<T>`	Применяет операцию к двум объектам типа `T` и возвращает результат тоже типа `T`. Его метод называется `apply()`
`Consumer<T>`	Применяет операцию к объекту типа `T`. Его метод называется `accept()`
`Supplier<T>`	Возвращает объект типа `T`. Его метод называется `get()`
`Function<T, R>`	Применяет операцию к объекту типа `T` и возвращает в качестве результата объект типа `R`. Его метод называется `apply()`
`Predicate<T>`	Выясняет, удовлетворяет ли объект типа `T` определенному ограничению. Возвращает булевское значение, указывающее на результат проверки. Его метод называется `test()`

ГЛАВА

16 Модули

В версии JDK 9 появилось новое и важное средство, называемое *модулями*. Модули предоставляют способ описания отношений и зависимостей кода, из которого состоит приложение. Модули также позволяют контролировать то, какие части модуля доступны другим модулям, а какие — нет. За счет использования модулей можно создавать более надежные и масштабируемые программы.

Как правило, модули наиболее полезны в крупных приложениях, поскольку они помогают сократить сложность управления, часто связанную с большой программной системой. Однако мелкие программы тоже выигрывают от применения модулей, потому что библиотека Java API теперь организована в виде модулей. Таким образом, теперь можно указывать, какие части Java API требуются вашей программе, а какие не нужны. Это позволяет развертывать программы с меньшим объемом пространства хранения, потребляемого во время выполнения, что особенно важно при создании кода, например, для небольших устройств, входящих в состав Интернета вещей (Internet of Things — IoT).

Поддержка модулей обеспечивается как языковыми элементами, в том числе несколькими ключевыми словами, так и улучшениями `javac`, `java` и других инструментов JDK. Кроме того, были предложены новые инструменты и форматы файлов. В результате JDK и исполняющая среда были существенно модернизированы с целью поддержки модулей. Короче говоря, модули являются важным дополнением и эволюционным шагом языка Java.

Основы модулей

В наиболее основополагающем смысле *модуль* представляет собой группу пакетов и ресурсов, на которые можно коллективно ссылаться по имени модуля. В *объявлении модуля* указывается имя модуля и определяется отношение модуля и его пакетов с другими модулями. Объявления модулей записываются в виде операторов в файле исходного кода Java и поддерживаются несколькими ключевыми словами, связанными с модулями:

```
exports          provides          transitive
module           requires          uses
open             to                with
opens
```

Важно понимать, что перечисленные выше ключевые слова распознаются *как ключевые слова* только в контексте объявления модуля. В других ситуациях они интерпретируются как идентификаторы. Таким образом, ключевое слово `module` можно было бы использовать в качестве имени параметра, хотя поступать так, безусловно, не рекомендуется. Тем не менее, создание контекстно-зависимых ключевых слов, связанных с модулем, предотвращает возникновение проблем с существующим кодом, в котором одно или несколько таких слов могут быть выбраны для идентификаторов.

Объявление модуля содержится в файле по имени `module-info.java`, т.е. модуль определяется в файле исходного кода Java.

Файл `module-info.java` затем компилируется с помощью `javac` в файл класса и известен как его *дескриптор модуля*. Файл `module-info.java` должен содержать только определение модуля, но не другие виды объявлений.

Объявление модуля начинается с ключевого слова `module`. Вот его общая форма:

```
module имя-модуля {
  // определение модуля
}
```

Имя модуля указывается в `имя-модуля` и обязано быть допустимым идентификатором Java или последовательностью идентификаторов, разделенных точками. Определение модуля находится в фигурных скобках. Хотя определение модуля может быть пустым (что приводит к объявлению, которое просто именует модуль), обычно в нем присутствует одна или несколько конструкций, устанавливающих характеристики модуля.

Простой пример модуля

В основе возможностей модуля лежат две ключевые особенности. Первая из них — способность модуля сообщать о том, что ему требуется другой модуль. Другими словами, один модуль может указывать, что он *зависит* от другого модуля. Отношение зависимости задается с помощью оператора `requires`. По умолчанию наличие необходимого модуля проверяется как на этапе компиляции, так и во время выполнения. Второй ключевой особенностью является способность модуля контролировать, какие из его пакетов доступны другому модулю, что достигается с помощью ключевого слова `exports`. Открытые и защищенные типы внутри пакета доступны другим модулям только в том случае, если они явно экспортированы. Здесь мы займемся примером, в котором демонстрируются обе возможности.

В следующем примере создается модульное приложение, в котором задействованы простые математические функции. Хотя это приложение специально сделано очень маленьким, оно иллюстрирует основные концепции и

процедуры, необходимые для создания, компиляции и запуска кода на основе модулей. Вдобавок показанный здесь общий подход применим и к более крупным, реальным приложениям. Настоятельно рекомендуется проработать пример на своем компьютере, внимательно следуя каждому шагу.

На заметку! В главе описан процесс создания, компиляции и запуска кода на основе модулей с помощью инструментов командной строки. Такой прием обладает двумя преимуществами. Во-первых, он подойдет всем программистам на Java, потому что не требует IDE-среды. Во-вторых, он очень четко иллюстрирует основы системы модулей, в том числе ее работу с каталогами. Вам придется вручную создать несколько каталогов и обеспечить размещение каждого файла в надлежащем каталоге. Как и следовало ожидать, при создании реальных приложений на основе модулей вы, по всей видимости, обнаружите, что легче пользоваться IDE-средой с поддержкой модулей, поскольку обычно она автоматизирует бо́льшую часть процесса. Однако изучение основ модулей с применением инструментов командной строки гарантирует, что вы обретете устойчивое понимание темы.

В приложении определяются два модуля. Первый модуль имеет имя `appstart` и содержит пакет `appstart.mymodappdemo`, устанавливающий точку входа приложения в классе `MyModAppDemo`. Таким образом, `MyModAppDemo` содержит метод `main()` приложения. Второй модуль называется `appfuncs` и содержит пакет `appfuncs.simplefuncs`, который включает класс `SimpleMathFuncs`. В классе `SimpleMathFuncs` определены три статических метода, реализующие ряд простых математических функций. Все приложение будет размещаться в дереве каталогов, которое начинается с `mymodapp`.

Прежде чем продолжить, уместно обсудить имена модулей. В последующих примерах имя модуля (скажем, `appfuncs`) является префиксом имени пакета (например, `appfuncs.simplefuncs`), который он содержит. Поступать так вовсе *не* обязательно, но такой прием используется в примере как способ четко указать, к какому модулю принадлежит пакет. Вообще говоря, при изучении и экспериментировании с модулями полезно выбирать короткие простые имена вроде тех, что применяются в настоящей главе, но вы вольны использовать любые удобные имена, которые вам нравятся. Тем не менее, при создании модулей, пригодных для распространения, вы должны соблюдать осторожность в отношении назначаемых имен, т.к. желательно, чтобы эти имена были уникальными. На момент написания книги рекомендовалось применять метод обратных доменных имен, при котором обратное имя домена, "владеющего" проектом, используется в качестве префикса для модуля. Скажем, в проекте, ассоциированном с `herbschildt.com`, для префикса модуля будет применяться `com.herbschildt`. (То же самое касается имен пакетов.) Поскольку соглашения об именовании со временем могут меняться, вам следует искать текущие рекомендации в документации по Java.

Итак, начнем. Первым делом создайте необходимые каталоги для исходного кода, выполнив перечисленные ниже шаги.

1. Создайте каталог по имени `mymodapp`. Он будет служить каталогом верхнего уровня для всего приложения.

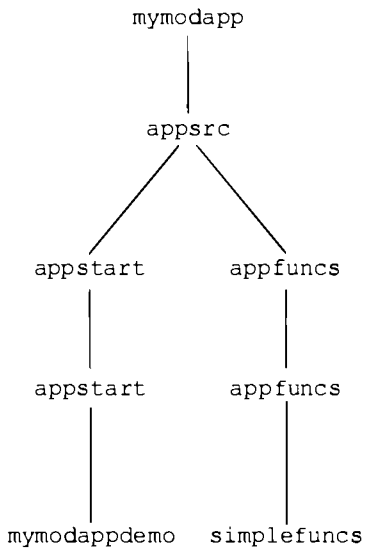
2. Внутри mymodapp создайте подкаталог под названием appsrc. Он послужит каталогом верхнего уровня для исходного кода приложения.
3. Внутри appsrc создайте подкаталог appstart, в нем — подкаталог с таким же именем appstart, а внутри него — подкаталог mymodappdemo. В результате вы получите следующее дерево каталогов, начинающееся с appsrc:

   ```
   appsrc\appstart\appstart\mymodappdemo
   ```

4. Внутри appsrc создайте подкаталог appfuncs, в нем — подкаталог с таким же именем appfuncs, а внутри него — подкаталог simplefuncs. В итоге вы получите следующее дерево каталогов, которое начинается с appsrc:

   ```
   appsrc\appfuncs\appfuncs\simplefuncs
   ```

Вот как должно выглядеть окончательное дерево каталогов:



После настройки каталогов можете заняться созданием файлов исходного кода приложения.

В текущем примере будут использоваться четыре файла исходного кода. В двух из них определяется приложение. Ниже приведено содержимое первого файла, SimpleMathFuncs.java. Обратите внимание, что класс SimpleMathFuncs находится в пакете appfuncs.simplefuncs.

```
// Простые математические функции.
package appfuncs.simplefuncs;
public class SimpleMathFuncs {
  // Выяснить, является ли a делителем b.
  public static boolean isFactor(int a, int b) {
    if((b%a) == 0) return true;
    return false;
  }
```

```
  // Возвратить наименьший положительный делитель, общий для a и b.
  public static int lcf(int a, int b) {
    // Разложить на множители, используя положительные значения.
    a = Math.abs(a);
    b = Math.abs(b);

    int min = a < b ? a : b;

    for(int i = 2; i <= min/2; i++) {
      if(isFactor(i, a) && isFactor(i, b))
        return i;
    }

    return 1;
  }

  // Возвратить наибольший положительный делитель, общий для a и b.
  public static int gcf(int a, int b) {
    // Разложить на множители, используя положительные значения.
    a = Math.abs(a);
    b = Math.abs(b);

    int min = a < b ? a : b;

    for(int i = min/2; i >= 2; i--) {
      if(isFactor(i, a) && isFactor(i, b))
        return i;
    }

    return 1;
  }
}
```

В классе SimpleMathFuncs определяются три простых статических метода, которые реализуют математические функции. Первый метод, isFactor(), возвращает true, если a является делителем b. Метод lcf() возвращает наименьший делитель, общий для a и b. Другими словами, он возвращает наименьший общий делитель для чисел a и b. Метод gcf() возвращает наибольший делитель, общий для a и b. В обоих случаях возвращается 1, когда общие делители не найдены. Файл SimpleMathFuncs.java должен быть помещен в следующий каталог:

```
appsrc\appfuncs\appfuncs\simplefuncs
```

Это каталог пакета appfuncs.simplefuncs.

Далее показано содержимое второго файла исходного кода, MyModAppDemo.java, в котором задействованы методы класса SimpleMathFuncs. Обратите внимание, что он находится в пакете appstart.mymodappdemo. Кроме того, в нем импортируется класс SimpleMathFuncs, т.к. в своей работе он полагается на SimpleMathFuncs.

```
// Простой пример приложения, основанного на модулях.

package appstart.mymodappdemo;

import appfuncs.simplefuncs.SimpleMathFuncs;

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

```
    if(SimpleMathFuncs.isFactor(2, 10))
      System.out.println("2 является делителем 10");

    System.out.println("Наименьший общий делитель для 35 и 105 равен " +
                       SimpleMathFuncs.lcf(35, 105));

    System.out.println("Наибольший общий делитель для 35 и 105 равен " +
                       SimpleMathFuncs.gcf(35, 105));

  }
}
```

Файл MyModAppDemo.java должен быть помещен в следующий каталог:

```
appsrc\appstart\appstart\mymodappdemo
```

Это каталог пакета appstart.mymodappdemo.

Затем для каждого модуля понадобится добавить файлы module-info.java, которые содержат определения модулей. Первым делом добавьте файл, определяющий модуль appfuncs:

```
// Определение модуля математических функций.
module appfuncs {
  // Экспортировать пакет appfuncs.simplefuncs.
  exports appfuncs.simplefuncs;
}
```

Как видите, модуль appfuncs экспортирует пакет appfuncs.simplefuncs, что делает его доступным для других модулей. Данный файл потребуется поместить в следующий каталог:

```
appsrc\appfuncs
```

Таким образом, он попадает в каталог модуля appfuncs, который находится выше каталогов пакетов.

Наконец, ниже представлено содержимое файла module-info.java для модуля appstart. Обратите внимание, что appstart требует модуля appfuncs.

```
// Определение модуля главного приложения.
module appstart {
  // Требует модуля appfuncs.
  requires appfuncs;
}
```

Этот файл должен быть помещен в каталог своего модуля:

```
appsrc\appstart
```

Прежде чем приступить к более подробным исследованиям операторов requires, exports и module, давайте скомпилируем и запустим код примера. Удостоверьтесь в том, что корректно создали каталоги и поместили каждый файл в соответствующий каталог.

Компиляция и запуск первого примера модуля

В версии JDK 9 компилятор javac был обновлен с целью поддержки модулей. Таким образом, подобно всем другим программам на Java програм-

мы на основе модулей компилируются с применением javac. Процесс прост, а основное отличие заключается в том, что обычно будет явно указываться *путь к модулю*, который сообщает компилятору местоположение для скомпилированных файлов. Прорабатывая пример, не забывайте о необходимости вводить команды javac в каталоге mymodapp, чтобы пути были правильными. Вспомните, что mymodapp является каталогом верхнего уровня для всего модульного приложения.

Первым делом скомпилируйте файл SimpleMathFuncs.java, используя следующую команду:

```
javac -d appmodules\appfuncs appsrc\appfuncs\appfuncs\simplefuncs
  \SimpleMathFuncs.java
```

Как уже упоминалось, команда должна вводиться в каталоге mymodapp. Обратите внимание на применение параметра -d, с помощью которого компилятору javac сообщается о том, куда поместить выходной файл .class. В примерах, рассматриваемых в главе, вершиной дерева каталогов для скомпилированного кода является appmodules. Приведенная выше команда по мере необходимости создает выходные каталоги пакетов для appfuncs.simplefuncs внутри appmodules\appfuncs.

А вот команда javac, которая компилирует файл module-info.java для модуля appfuncs:

```
javac -d appmodules\appfuncs appsrc\appfuncs\module-info.java
```

Она помещает файл module-info.class в каталог appmodules\appfuncs.

Хотя предыдущий двухэтапный процесс работает, он был показан в первую очередь ради обсуждения. Обычно компилировать файл module-info.java и файлы исходного кода модуля проще в одной командной строке. Ниже две предшествующих команды javac объединены в одну:

```
javac -d appmodules\appfuncs appsrc\appfuncs\module-info.java
  appsrc\appfuncs\appfuncs\simplefuncs\SimpleMathFuncs.java
```

Команда обеспечивает помещение каждого скомпилированного файла в соответствующий каталог модуля или пакета.

Теперь скомпилируйте файлы module-info.java и MyModAppDemo.java для модуля appstart:

```
javac --module-path appmodules -d appmodules\appstart
  appsrc\appstart\module-info.java
  appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java
```

Обратите внимание на наличие параметра --module-path. Он задает путь к модулю, т.е. путь, который компилятор будет просматривать в поисках пользовательских модулей, затребованных в файле module-info.java. В этом случае компилятор будет искать модуль appfuncs, потому что он нужен модулю appstart. Кроме того, выходной каталог в команде указан как appmodules\appstart. Таким образом, файл module-info.class будет располагаться в каталоге модуля appmodules\appstart, а MyModAppDemo.class — в каталоге пакета appmodules\appstart\appstart\mymodappdemo.

Завершив компиляцию, можете запустить приложение с помощью команды `java`:

```
java --module-path appmodules -m appstart/appstart.mymodappdemo.MyModAppDemo
```

В параметре `--module-path` задан путь к модулям приложения. Как упоминалось ранее, `appmodules` представляет собой каталог в верхней части дерева скомпилированных модулей. В параметре `-m` указан класс, содержащий точку входа приложения, и в рассматриваемом случае это имя класса, содержащего метод `main()`. В результате запуска программы вы увидите следующий вывод:

```
2 является делителем 10
Наименьший общий делитель для 35 и 105 равен 5
Наибольший общий делитель для 35 и 105 равен 7
```

Более подробный анализ операторов `requires` и `exports`

Предыдущий пример приложения, основанного на модулях, опирался на две основополагающих особенности модульной системы: возможность установления зависимости и способность ее удовлетворения. Такие возможности указываются посредством операторов `requires` и `exports` в объявлении модуля. Каждый заслуживает более пристального изучения.

Вот форма оператора `requires`, используемая в примере:

```
requires moduleName;
```

Здесь в `moduleName` задано имя модуля, который требуется модулю, в котором встречается оператор `requires`. Это означает, что для успешной компиляции текущего модуля требуемый модуль должен существовать. На языке модулей говорят, что текущий модуль *читает* модуль, указанный в операторе `requires`. Когда есть потребность в нескольких модулях, они должны задаваться в собственных операторах `requires`. Таким образом, объявление модуля может содержать набор различных операторов `requires`. В общем случае оператор `requires` позволяет гарантировать, что программа имеет доступ к модулям, в которых она нуждается.

Ниже показана общая форма оператора `exports`, применяемая в примере:

```
exports packageName;
```

В `packageName` указывается имя пакета, экспортируемого модулем, в котором находится данный оператор. Модулю разрешено экспортировать столько пакетов, сколько необходимо, причем каждый из них задается в отдельном операторе `exports`. В итоге модуль может иметь несколько операторов `exports`.

Когда модуль экспортирует пакет, он делает все открытые и защищенные типы в пакете доступными другим модулям. Кроме того, открытые и защищенные члены этих типов тоже доступны. Однако если пакет внутри модуля не экспортируется, то он является закрытым для этого модуля, включая все

его открытые типы. Скажем, даже если класс объявлен как public в пакете, но пакет не экспортируется явно оператором exports, то такой класс не будет доступен остальным модулям. Важно понимать, что открытые и защищенные типы пакета вне зависимости от того, экспортированы они или нет, всегда доступны внутри модуля данного пакета. Оператор exports просто делает их доступными внешним модулям. Таким образом, любой неэкспортированный пакет предназначен только для внутреннего потребления модулем.

Ключевой аспект понимания операторов requires и exports в том, что они работают вместе. Если один модуль зависит от другого, то такую зависимость необходимо указать с помощью requires. Модуль, от которого зависит другой модуль, обязан явно экспортировать (т.е. делать доступными) пакеты, в которых нуждается зависимый модуль. Если любая сторона этого отношения зависимости отсутствует, тогда зависимый модуль компилироваться не будет. Как и в предыдущем примере, MyModAppDemo использует функции из SimpleMathFuncs. В результате объявление модуля appstart содержит оператор requires, в котором указан модуль appfuncs. Объявление модуля appfuncs экспортирует пакет appfuncs.simplefuncs, делая доступными открытые типы в классе SimpleMathFuncs. Поскольку обе стороны отношения зависимости удовлетворены, приложение может быть скомпилировано и запущено. Если одна из них отсутствует, тогда компиляция потерпит неудачу.

Важно подчеркнуть, что операторы requires и exports должны встречаться только внутри оператора модуля. Кроме того, оператор модуля сам по себе должен находиться в файле по имени module-info.java.

Модуль java.base и модули платформы

Как упоминалось ранее в главе, начиная с версии JDK 9, пакеты Java API были встроены в модули. На самом деле модульность Java API является одним из основных преимуществ, которые обеспечиваются появлением модулей. Из-за своей особой роли модули Java API называются *модулями платформы*, и все их имена начинаются с префикса java, например, java.base, java.desktop и java.xml. Модульность Java API позволяет развертывать приложение только с теми пакетами, которые ему необходимы, а не со всей средой JRE. Из-за размера полной версии JRE это очень важное улучшение.

Тот факт, что все пакеты библиотеки Java API теперь находятся в модулях, вызывает следующий вопрос: как метод main() в MyModAppDemo из предыдущего примера может вызывать System.out.println() без оператора requires для модуля, который содержит класс System? Очевидно, что программа не скомпилируется и не запустится, если отсутствует System. Аналогичный вопрос относится и к работе с классом Math в SimpleMathFuncs. Ответ на вопрос кроется в java.base.

Наиболее важным из модулей платформы является java.base. Он включает и экспортирует помимо множества других такие фундаментальные для Java пакеты, как java.lang, java.io и java.util. По причине своей важности пакет java.base *автоматически доступен* всем модулям. Вдобавок

все остальные модули автоматически требуют java.base. Нет никакой необходимости помещать оператор requires java.base в объявление модуля. (Интересно отметить, что в явном указании java.base нет ничего плохого, просто это необязательно.) Таким образом, почти так же, как пакет java.lang автоматически доступен во всех программах без применения оператора import, модуль java.base будет автоматически доступным для всех программ на основе модулей без явного запроса.

Поскольку java.base включает пакет java.lang, а java.lang содержит класс System, класс MyModAppDemo в предыдущем примере может автоматически использовать System.out.println() без явного оператора requires. То же самое относится и к классу Math в SimpleMathFuncs, потому что Math тоже находится в java.lang. Приступив к созданию собственных модульных приложений, вы заметите, что многие классы Java API, которые обычно нужны, расположены в пакетах, входящих в состав java.base. Таким образом, автоматическое включение java.base упрощает создание кода на основе модулей, т.к. основные пакеты Java доступны автоматически.

И последнее: начиная с JDK 9, в документации по Java API теперь указывается имя модуля, где содержится пакет. Если речь идет о модуле java.base, тогда вы можете напрямую потреблять содержимое этого пакета. В противном случае объявление модуля должно включать оператор requires для желаемого модуля.

Унаследованный код и неименованные модули

При работе с первым примером модульной программы у вас мог возникнуть еще один вопрос. Поскольку в Java теперь поддерживаются модули, а пакеты Java API также содержатся в модулях, почему все остальные программы из предыдущих глав компилируются и выполняются без ошибок, даже если модули в них не применяются? Выразимся в более общем плане: учитывая то, что код на Java существует уже свыше 20 лет и (на момент написания книги) в подавляющем большинстве кода модули не используются, то как удается компилировать, запускать и поддерживать устаревший код с помощью компилятора JDK 9 или его более поздней версии? С учетом изначальной философии Java, т.е. "написанное однажды выполняется везде, в любое время, всегда", вопрос крайне важен, т.к. необходимо поддерживать обратную совместимость. Вы увидите, что язык Java отвечает на данный вопрос, предлагая элегантные и почти прозрачные средства обеспечения обратной совместимости с написанным ранее кодом.

Для поддержки унаследованного кода предназначены два ключевых средства. Одно из них — *неименованный модуль*. Когда вы используете код, не являющийся частью именованного модуля, он автоматически становится частью неименованного модуля, который обладает двумя важными характеристиками. Во-первых, все пакеты в неименованном модуле автоматически экспортируются. Во-вторых, неименованный модуль имеет доступ абсолютно ко всем остальным модулям. Таким образом, когда модули в программе не

применяются, все модули Java API платформы Java автоматически доступны через неименованный модуль.

Вторым ключевым средством, поддерживающим унаследованный код, следует считать автоматическое использование пути к классу, а не пути к модулю. При компиляции программы, в которой модули не задействованы, применяется механизм пути к классам, как было со времен первоначального выпуска Java. В результате программа компилируется и запускается в той же манере, как было до появления модулей.

Из-за неименованного модуля и автоматического использования пути к классам в показанных ранее примерах программ не было необходимости объявлять какие-либо модули. Примеры корректно запускаются вне зависимости от того, компилируется они современным или же компилятором предшествующей версии наподобие JDK 8. Таким образом, несмотря на то, что модули являются средством, которое оказывает значительное влияние на Java, совместимость с устаревшим кодом сохраняется. Данный подход также обеспечивает плавный, ненавязчивый и неразрушающий путь перехода к модулям. В итоге он позволяет переносить унаследованное приложение в модули в приемлемом темпе. Кроме того, появляется возможность избежать применения модулей, когда они не нужны.

Прежде чем двигаться дальше, необходимо сделать одно важное замечание. Для примеров программ, используемых в других местах книги, и для примеров программ в целом использование модулей не приносит какой-либо пользы. Модульность в таких примерах просто добавила бы беспорядка и усложнила бы их без особой на то причины или выгоды. Вдобавок многие простые программы нет нужды помещать в модули. По причинам, изложенным в начале главы, модули часто приносят наибольшую пользу при создании коммерческих программ и потому ни в каких примерах за рамками текущей главы модули не применяются. Это также позволяет компилировать примеры и запускать их в среде, предшествующей JDK 9, что важно для читателей, использующих более раннюю версию Java. Таким образом, примеры программ, рассмотренные в книге, за исключением тех, что приводятся в настоящей главе, работают для JDK, вышедших до и после появления модулей.

Экспортирование в конкретный модуль

Базовая форма оператора `exports` открывает доступ к пакету любым другим модулям. Часто именно это и нужно. Тем не менее, в ряде специализированных ситуаций при разработке может возникнуть необходимость сделать пакет доступным только *конкретному набору* модулей, а не *всем* другим модулям. Скажем, разработчик библиотеки может решить экспортировать пакет поддержки в несколько других модулей внутри библиотеки, но не делать его доступным для общего потребления. Достичь желаемого можно, добавив конструкцию `to` к оператору `exports`.

В конструкции `to` оператора `exports` указывается список из одного или нескольких модулей, которые имеют доступ к экспортируемому пакету. Более

того, доступ будет предоставлен только тем модулям, имена которых перечислены в конструкции to. На языке модулей конструкция to создает так называемый *уточненный экспорт*.

Вот форма оператора exports с конструкцией to:

```
exports packageName to moduleNames;
```

Здесь packageName представляет собой разделяемый запятыми список модулей, которым выдается доступ к экспортирующему модулю.

Можете опробовать конструкцию to, изменив содержимое файла module-info.java для модуля appfuncs, как показано ниже:

```
// Определение модуля, в котором используется конструкция to.
module appfuncs {
  // Экспортировать пакет appfuncs.simplefuncs в appstart.
  exports appfuncs.simplefuncs to appstart;
}
```

Теперь simplefuncs экспортируется только в appstart и ни в какие другие модули. После внесения такого изменения перекомпилируйте приложение с помощью следующей команды javac:

```
javac -d appmodules --module-source-path appsrc
  appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java
```

После компиляции запустите приложение, как объяснялось ранее.

В данном примере также задействовано еще одно средство, связанное с модулями. Взгляните внимательно на предыдущую команду javac. Первым делом обратите внимание на наличие параметра --module-source-path. Исходный путь к модулю задает вершину дерева каталогов с исходным кодом модуля. Параметр --module-source-path обеспечивает автоматическую компиляцию файлов в дереве ниже указанного каталога, которым является appsrc. Параметр --module-source-path должен применяться вместе с параметром -d, чтобы гарантировать сохранение скомпилированных модулей в надлежащих каталогах внутри appmodules. В случае ввода показанной выше формы команды компилятор javac будет работать в *многомодульном режиме*, позволяя одновременно компилировать более одного модуля. Многомодульный режим компиляции здесь особенно полезен, т.к. конструкция to относится к конкретному модулю, а затребованный модуль должен иметь доступ к экспортируемому пакету. Таким образом, во избежание выдачи сообщений с предупреждениями и/или ошибками на этапе компиляции в рассматриваемом случае необходимы и appstart, и appfuncs. Многомодульный режим позволяет избежать этой проблемы, потому что оба модуля компилируются одновременно.

Многомодульный режим javac обладает еще одним преимуществом. Он автоматически отыскивает и компилирует все исходные файлы для приложения, создавая необходимые выходные каталоги. Из-за преимуществ, которые предлагает многомодульный режим компиляции, он будет задействован и в последующих примерах.

На заметку! Как правило, уточненный экспорт является особым случаем. Чаще всего ваши модули будут либо предоставлять неуточненный экспорт пакета, либо не экспортировать пакет вообще, оставляя его недоступным. Таким образом, уточненный экспорт обсуждается главным образом ради полноты картины. Кроме того, уточненный экспорт сам по себе не предотвращает неправомерное использование экспортированного пакета вредоносным кодом в модуле, который маскируется под целевой модуль. Обсуждение приемов безопасности, необходимых для предотвращения этого, выходят за рамки материала книги. Подробную информацию о безопасности в данном отношении и общие сведения о безопасности Java ищите в документации Oracle.

Использование `requires transitive`

Рассмотрим ситуацию, когда есть три модуля, A, B и C, со следующими зависимостями:

A требует B;
B требует C.

В такой ситуации совершенно ясно, что поскольку A зависит от B, а B зависит от C, модуль A имеет косвенную зависимость от C. Пока в модуле A не применяется напрямую какое-то содержимое модуля C, можно в файле `module-info.java` для A просто затребовать B и в файле `module-info.java` для B экспортировать пакеты, требующиеся в A:

```
// Файл module-info.java для A.
module A {
  requires B;
}
// Файл module-info.java для B.
module B {
  exports packageName;
  requires C;
}
```

Здесь `packageName` представляет собой заполнитель для пакета, экспортируемого модулем B и используемого в модуле A. Хотя такой подход работает при условии, что в A не нужно задействовать какой-нибудь тип, определенный в C, если в A понадобится получить доступ к типу из модуля C, тогда возникнет проблема. Решить ее можно двумя способами.

Первое решение предусматривает добавление в файл `module-info.java` для A оператора `requires C`:

```
//Файл module-info.java для A,модифицированный с целью явного затребования C
module A {
  requires B;
  requires C;  // также затребовать C
}
```

Безусловно, решение работоспособно, но если модуль B будет потребляться многими модулями, то оператор `require C` придется добавить ко всем определениям модулей, которым требуется B, что в равной степени и утомительно, и чревато ошибками. К счастью, существует более эффективное ре-

шение — создать *подразумеваемую зависимость* от C, которая также называется *подразумеваемой читаемостью*.

Чтобы создать подразумеваемую зависимость, добавьте ключевое слово transitive после конструкции requires, требующей модуль, для которого необходима подразумеваемая читаемость. В данном примере нужно изменить файл module-info.java для B:

```
// Файл module-info.java для B.
module B {
  exports packageName;
  requires transitive C;
}
```

Теперь модуль C затребован как транзитивный. После внесения такого изменения любой модуль, который зависит от B, будет также автоматически зависеть от C. В итоге модуль A получит доступ к C.

Можете поэкспериментировать с оператором requires transitive, переделав предыдущий пример модульного приложения, чтобы удалить метод isFactor() из класса SimpleMathFuncs в пакете appfuncs.simplefuncs и поместить его в новый класс, модуль и пакет. Новый класс получит имя SupportFuncs, модуль — имя appsupport, а пакет — имя appsupport.supportfuncs. Затем модуль appfuncs добавит зависимость от модуля appsupport с помощью requires transitive, что позволит модулям appfuncs и appstart получить к нему доступ без необходимости иметь в appstart собственный оператор requires. Прием работает, поскольку appstart получает к нему доступ через оператор requires transitive в appfuncs. Ниже весь процесс описан более подробно.

Первым делом понадобится создать каталоги для исходного кода, которые поддерживают новый модуль appsupport. Сначала создайте внутри каталога appsrc каталог модуля appsupport, предназначенный для вспомогательных функций. Внутри appsupport создайте каталог пакета, добавив подкаталог appsupport, а затем подкаталог supportfuncs. Дерево каталогов для appsupport должно выглядеть так:

```
appsrc\appsupport\appsupport\supportfuncs
```

После каталогов создайте класс SupportFuncs. Обратите внимание, что SupportFuncs входит в состав пакета appsupport.supportfuncs, поэтому вы должны поместить его в каталог пакета appsupport.supportfuncs.

```
// Вспомогательные функции.
package appsupport.supportfuncs;
public class SupportFuncs {
  // Выяснить, является ли a делителем b.
  public static boolean isFactor(int a, int b) {
    if((b%a) == 0) return true;
    return false;
  }
}
```

Обратите внимание, что метод isFactor() теперь является частью класса SupportFuncs, а не SimpleMathFuncs.

Далее создайте в каталоге appsrc\appsupport файл module-info.java для модуля appsupport:

```
// Определение модуля для appsupport.
module appsupport {
  exports appsupport.supportfuncs;
}
```

Как видите, он экспортирует пакет appsupport.supportfuncs.

Поскольку метод isFactor() определен в классе Supportfuncs, удалите его из SimpleMathFuncs. Таким образом, содержимое SimpleMathFuncs.java приобретает следующий вид:

```
// Простые математические функции; метод isFactor() удален.

package appfuncs.simplefuncs;

import appsupport.supportfuncs.SupportFuncs;

public class SimpleMathFuncs {

  // Возвратить наименьший положительный делитель, общий для a и b.
  public static int lcf(int a, int b) {
    // Разложить на множители, используя положительные значения.
    a = Math.abs(a);
    b = Math.abs(b);

    int min = a < b ? a : b;

    for(int i = 2; i <= min/2; i++) {
      if(SupportFuncs.isFactor(i, a) && SupportFuncs.isFactor(i, b))
        return i;
    }

    return 1;
  }

  // Возвратить наибольший положительный делитель, общий для a и b.
  public static int gcf(int a, int b) {
    // Разложить на множители, используя положительные значения.
    a = Math.abs(a);
    b = Math.abs(b);

    int min = a < b ? a : b;

    for(int i = min/2; i >= 2; i--) {
      if(SupportFuncs.isFactor(i, a) && SupportFuncs.isFactor(i, b))
        return i;
    }

    return 1;
  }
}
```

Обратите внимание, что теперь класс SupportFuncs импортирован, а вызовы isFactor() ссылаются на имя класса SupportFuncs.

Модифицируйте файл module-info.java для appfuncs, чтобы в его операторе requires модуль appsupport был помечен как transitive:

```
// Определение модуля для appfuncs.
module appfuncs {
  // Экспортировать пакет appfuncs.simplefuncs.
  exports appfuncs.simplefuncs;

  // Затребовать модуль appsupport и сделать его транзитивным.
  requires transitive appsupport;
}
```

Поскольку в appfuncs модуль appsupport затребован как транзитивный, нет никакой необходимости в наличии файла module-info.java для appstart. Подразумевается его зависимость от appsupport. Таким образом, изменять содержимое файла module-info.java для запуска приложения не нужно.

В заключение модифицируйте содержимое файла MyModAppDemo.java, чтобы отразить эти изменения. В частности, теперь в нем должен импортироваться класс SupportFuncs и указываться при вызове метода isFactor():

```
// Обновление с целью использования класса SupportFuncs.
package appstart.mymodappdemo;

import appfuncs.simplefuncs.SimpleMathFuncs;
import appsupport.supportfuncs.SupportFuncs;

public class MyModAppDemo {
  public static void main(String[] args) {

    // Теперь ссылка на метод isFactor() производится
    // через SupportFuncs, а не SimpleMathFuncs.
    if(SupportFuncs.isFactor(2, 10))
      System.out.println("2 является делителем 10");

    System.out.println("Наименьший общий делитель для 35 и 105 равен " +
                       SimpleMathFuncs.lcf(35, 105));
    System.out.println("Наибольший общий делитель для 35 и 105 равен " +
                       SimpleMathFuncs.gcf(35, 105));
  }
}
```

После выполнения всех предыдущих шагов заново скомпилируйте программу с применением команды javac в многомодульном режиме:

```
javac -d appmodules --module-source-path appsrc
  appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java
```

Как объяснялось ранее, при многомодульном режиме компиляции внутри каталога appmodules автоматически создаются подкаталоги для модулей. Запустить программу можно с помощью следующей команды:

```
java --module-path appmodules -m appstart/appstart.mymodappdemo.MyModAppDemo
```

Программа произведет тот же вывод, что и предыдущая версия, но на этот раз будут затребованы три разных модуля.

Чтобы удостовериться в том, что модификатор transitive действительно необходим приложению, удалите его из файла module-info.java для appfuncs и попробуйте скомпилировать программу. Возникнет ошибка, потому что модуль appsupport больше не будет доступным модулю appstart.

Проведите еще один эксперимент. В файле module-info.java для appsupport попробуйте экспортировать пакет appsupport.supportfuncs только в appfuncs, используя уточненный экспорт:

```
exports appsupport.supportfuncs to appfuncs;
```

Попытайтесь скомпилировать программу. Как видите, программа не компилируется, поскольку теперь вспомогательный метод isFactor() оказывается недоступным классу MyModAppDemo, находящемуся в модуле appstart. Согласно приведенным ранее объяснениям уточненный экспорт ограничивает доступ к пакету только теми модулями, которые указаны в конструкции to.

И последнее замечание, которое касается особого исключения в синтаксисе языка Java, связанного с оператором requires: если сразу после transitive следует разделитель (например, точка с запятой), то transitive интерпретируется как идентификатор (скажем, имя модуля), а не как ключевое слово.

Использование служб

В программировании часто бывает полезно отделять то, *что* должно быть сделано, от того, *как* оно делается. В главе 9 вы узнали, что один из способов достижения такой цели в языке Java предусматривает применение интерфейсов. Интерфейс определяет *что*, а реализующий его класс — *как*. Концепцию можно расширить так, чтобы реализующий класс предоставлялся кодом, который находится за пределами программы, за счет использования *подключаемого модуля*. При таком подходе возможности приложения можно совершенствовать, модернизировать либо изменять, просто меняя подключаемый модуль. Ядро приложения остается незатронутым. Архитектура приложений с подключаемыми модулями поддерживается в Java через применение *служб* и *поставщиков служб*. По причинам их крайней важности, особенно для крупных коммерческих приложений, их поддержка обеспечивается системой модулей Java.

Прежде чем начать, необходимо отметить, что приложения, в которых используются службы и поставщики служб, обычно довольно сложны. Следовательно, вы можете обнаруживать, что средства модулей на основе служб требуются нечасто. Однако поскольку поддержка служб составляет довольно значительную часть системы модулей, важно иметь общее представление о том, как работают эти средства. Кроме того, далее предложен простой пример, который иллюстрирует основные приемы их применения.

Основы служб и поставщиков служб

Служба в Java представляет собой программную единицу, функциональность которой определяется интерфейсом или абстрактным классом. Таким образом, служба задает в общем виде некоторую форму программной деятельности. Конкретная реализация службы предоставляется *поставщиком служб*. Другими словами, служба определяет форму какого-то действия, а поставщик служб предоставляет это действие.

Как уже упоминалось, службы часто используются для поддержки подключаемой архитектуры. Например, служба может применяться для поддержки перевода с одного языка на другой. В таком случае служба поддерживает перевод в целом. Поставщик служб предоставляет конкретный перевод, скажем, с немецкого языка на английский или с французского языка на китайский. Из-за того, что все поставщики служб реализуют один и тот же интерфейс, появляется возможность использовать разные переводчики для перевода на разные языки без необходимости внесения изменений в ядро приложения. Можно просто сменить поставщика служб.

Поставщики служб поддерживаются обобщенным классом `ServiceLoader`, который доступен в пакете `java.util` и объявлен следующим образом:

```
class ServiceLoader<S>
```

В `S` задается тип службы. Поставщики служб загружаются с помощью метода `load()`. Он имеет несколько форм; ниже показана форма, которая будет применяться в главе:

```
public static <S> ServiceLoader<S> load(Class <S> serviceType)
```

В `serviceType` указывается объект `Class` для желаемого типа службы. Вспомните, что в объекте `Class` инкапсулирована информация о классе. Получить экземпляр `Class` можно многими способами, один из которых связан с использованием литерала класса. В качестве напоминания ниже показана общая форма литерала класса:

имя-класса`.class`

В имя-класса задается имя нужного класса.

В результате вызова метода `load()` возвращается экземпляр `ServiceLoader` для приложения, который поддерживает итерацию и может применяться в цикле `for` стиля "for-each". Таким образом, чтобы отыскать конкретного провайдера, просто ищите его с помощью цикла.

Ключевые слова, связанные со службами

Модули поддерживают службы через использование ключевых слов `provides`, `uses` и `with`. Модуль указывает, что он предоставляет службу, с помощью оператора `provides`. Модуль отражает тот факт, что ему требуется служба, посредством оператора `uses`. Специфический тип поставщика служб объявляется с применением `with`. Когда они используются все вместе, то позволяют задать модуль, предоставляющий службу, модуль, который нуждается в этой службе, и конкретную реализацию самой службы. Кроме того, система модулей гарантирует, что службы и поставщики служб доступны и будут найдены.

Вот общая форма оператора `provides`:

```
provides serviceType with implementationTypes;
```

В `serviceType` задается тип службы, которая часто является интерфейсом, хотя также применяются и абстрактные классы. Список типов реализации,

разделенных запятыми, помещается в implementationTypes. Следовательно, для предоставления службы в модуле указывается как имя службы, так и ее реализация.

Ниже приведена общая форма оператора uses:

```
uses serviceType;
```

В serviceType указывается тип требующейся службы.

Пример службы, основанной на модулях

Чтобы продемонстрировать работу со службами, мы добавим службу в разрабатываемый пример модульного приложения. Для простоты начнем с первой версии приложения, показанной в начале главы, и добавим в нее два новых модуля. Первый называется userfuncs. В нем будут определены интерфейсы с поддержкой функций, выполняющих бинарные операции, в которых каждый аргумент и результат имеют тип int. Второй модуль называется userfuncsimp и содержит конкретные реализации интерфейсов.

Первым делом создайте необходимые каталоги для исходного кода.

1. Добавьте в каталог appsrc подкаталоги с именами userfuncs и userfuncsimp.
2. Добавьте в каталог userfuncs подкаталог с тем же самым именем userfuncs, а в него — подкаталог binaryfuncs. Таким образом, вы получите следующее дерево каталогов, начинающееся с appsrc:

   ```
   appsrc\userfuncs\userfuncs\binaryfuncs
   ```

3. Добавьте в каталог userfuncsimp подкаталог с тем же самым именем userfuncsimp, а в него — подкаталог binaryfuncsimp. В итоге вы получите следующее дерево каталогов, начинающееся с appsrc:

   ```
   appsrc\userfuncsimp\userfuncsimp\binaryfuncsimp
   ```

Текущий пример расширяет первоначальную версию приложения, обеспечивая поддержку других функций вдобавок к тем, которые встроены в приложение. Вспомните, что класс SimpleMathFuncs предоставляет три встроенные функции: isFactor(), lcf() и gcf(). Хотя в него можно было бы добавить дополнительные функции, для этого пришлось бы модифицировать и заново компилировать приложение. Благодаря внедрению служб появляется возможность "подключать" новые функции во время выполнения, не изменяя приложение, что и будет делаться в рассматриваемом примере.

В данном случае служба предоставляет функции, которые принимают два аргумента типа int и возвращают результат типа int. Конечно, можно поддерживать и другие типы функций, если определить дополнительные интерфейсы, но поддержки бинарных целочисленных функций вполне достаточно для преследуемых целей, кроме того, размер исходного кода примера остается управляемым.

Интерфейсы служб

Понадобятся два интерфейса, связанных со службами. Один задает форму действия, а другой указывает форму поставщика этого действия. Определения обоих интерфейсов находятся в каталоге binaryfuncs и оба они входят в состав пакета userfuncs.binaryfuncs. Первый интерфейс по имени BinaryFunc объявляет форму бинарной функции:

```
// В этом интерфейсе определен метод, который принимает два аргумента
// типа int и возвращает результат типа int. Таким образом, он может
// описывать любую бинарную операцию с двумя целочисленными значениями,
// которая возвращает целое число.

package userfuncs.binaryfuncs;

public interface BinaryFunc {
  // Получить имя функции.
  public String getName();

  // Это функция, подлежащая выполнению. Она будет
  // предоставляться конкретными реализациями.
  public int func(int a, int b);
}
```

Интерфейс BinaryFunc объявляет форму объекта, который может реализовывать бинарную целочисленную функцию, что указывается методом func(). С помощью метода getName() можно получить имя функции, которое будет использоваться для выяснения типа реализуемой функции. Интерфейс BinaryFunc реализуется классом, предоставляющим бинарную функцию.

Второй интерфейс объявляет форму поставщика служб и называется BinFuncProvider:

```
// Этот интерфейс определяет форму поставщика служб,
// который получает экземпляры BinaryFunc.
package userfuncs.binaryfuncs;

import userfuncs.binaryfuncs.BinaryFunc;

public interface BinFuncProvider {
  // Получить экземпляр BinaryFunc.
  public BinaryFunc get();
}
```

В интерфейсе BinFuncProvider объявлен только один метод get(), который применяется для получения экземпляра BinaryFunc. Этот интерфейс должен быть реализован классом, желающим предоставлять экземпляры BinaryFunc.

Классы реализации

В рассматриваемом примере поддерживаются две конкретные реализации BinaryFunc. Первая реализация, AbsPlus, возвращает сумму абсолютных значений своих аргументов. Вторая реализация, AbsMinus, возвращает результат

вычитания абсолютного значения второго аргумента из абсолютного значения первого аргумента. Они предоставляются классами AbsPlusProvider и AbsMinusProvider. Исходный код этих классов должен храниться в каталоге binaryfuncsimp и принадлежать пакету userfuncsimp.binaryfuncsimp. Вот код для AbsPlus:

```
// AbsPlus предоставляет конкретную реализацию BinaryFunc,
// которая возвращает результат abs(a) + abs(b).
package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.BinaryFunc;

public class AbsPlus implements BinaryFunc {
  // Возвратить имя этой функции.
  public String getName() {
    return "absPlus";
  }

  // Реализовать функцию AbsPlus.
  public int func(int a, int b) { return Math.abs(a) + Math.abs(b); }
}
```

Класс AbsPlus реализует метод func() таким образом, что он возвращает результат сложения абсолютных значений a и b. Обратите внимание, что getName() возвращает строку "absPlus", идентифицирующую эту функцию.

Ниже приведен код AbsMinus:

```
// AbsMinus предоставляет конкретную реализацию BinaryFunc,
// которая возвращает результат abs(a) - abs(b).
package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.BinaryFunc;

public class AbsMinus implements BinaryFunc {
  // Возвратить имя этой функции.
  public String getName() {
    return "absMinus";
  }

  // Реализовать функцию AbsMinus.
  public int func(int a, int b) { return Math.abs(a) - Math.abs(b); }
}
```

Метод func() реализован для возвращения разности абсолютных значений a и b, а метод getName() возвращает строку "absMinus".

Для получения экземпляра AbsPlus используется класс AbsPlusProvider. Он реализует интерфейс BinFuncProvider и выглядит следующим образом:

```
// Поставщик для функции AbsPlus.

package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.*;

public class AbsPlusProvider implements BinFuncProvider {

  // Предоставить объект AbsPlus.
  public BinaryFunc get() { return new AbsPlus(); }
}
```

Метод get() просто возвращает новый объект AbsPlus. Хотя данный поставщик очень прост, важно понимать, что некоторые поставщики служб будут намного сложнее.

Поставщик для AbsMinus называется AbsMinusProvider и показан ниже:

```
// Поставщик для функции AbsMinus.
package userfuncsimp.binaryfuncsimp;
import userfuncs.binaryfuncs.*;
public class AbsMinusProvider implements BinFuncProvider {
  // Предоставить объект AbsMinus.
  public BinaryFunc get() { return new AbsMinus(); }
}
```

Метод get() поставщика возвращает объект AbsMinus.

Файлы определения модулей

Далее необходимы два файла определения модулей. Первый предназначен для модуля userfuncs и имеет такое содержимое:

```
module userfuncs {
  exports userfuncs.binaryfuncs;
}
```

Представленный выше код должен быть помещен в файл module-info.java, находящийся в каталоге модуля userfuncs. Обратите внимание, что в нем экспортируется пакет userfuncs.binaryfuncs, где определены интерфейсы BinaryFunc и BinFuncProvider.

Вот как выглядит содержимое второго файла module-info.java с определением модуля, содержащего реализации. Он должен быть помещен в каталог модуля userfuncsimp.

```
module userfuncsimp {
  requires userfuncs;
  provides userfuncs.binaryfuncs.BinFuncProvider with
    userfuncsimp.binaryfuncsimp.AbsPlusProvider,
    userfuncsimp.binaryfuncsimp.AbsMinusProvider;
}
```

В модуле userfuncsimp требуется модуль userfuncs, поскольку последний содержит интерфейсы BinaryFunc и BinFuncProvider, которые нужны реализациям. Модуль userfuncsimp предоставляет реализации BinFuncProvider в виде классов AbsPlusProvider и AbsMinusProvider.

Демонстрация поставщиков служб в классе MyModAppDemo

Чтобы продемонстрировать работу со службами, метод main() класса MyModAppDemo расширен с целью применения AbsPlus и AbsMinus, для чего загружает их во время выполнения с помощью ServiceLoader.load(). Вот модифицированный код:

```
// Модульное приложение, демонстрирующее использование служб
// и поставщиков служб.

package appstart.mymodappdemo;

import java.util.ServiceLoader;

import appfuncs.simplefuncs.SimpleMathFuncs;
import userfuncs.binaryfuncs.*;

public class MyModAppDemo {
  public static void main(String[] args) {

    // Первым делом использовать встроенные функции, как и ранее.
    if(SimpleMathFuncs.isFactor(2, 10))
      System.out.println("2 является делителем 10");

    System.out.println("Наименьший общий делитель для 35 и 105 равен " +
      SimpleMathFuncs.lcf(35, 105));

    System.out.println("Наибольший общий делитель для 35 и 105 равен " +
      SimpleMathFuncs.gcf(35, 105));

    // Теперь использовать пользовательские операции, основанные на службах

    // Получить загрузчик служб для бинарных функций.
    ServiceLoader<BinFuncProvider> ldr =
      ServiceLoader.load(BinFuncProvider.class);

    BinaryFunc binOp = null;

    // Найти поставщика для absPlus и получить функцию.
    for(BinFuncProvider bfp : ldr) {
      if(bfp.get().getName().equals("absPlus")) {
        binOp = bfp.get();
        break;
      }
    }

    if(binOp != null)
      System.out.println("Результат выполнения функции absPlus: " +
                         binOp.func(12, -4));
    else
      System.out.println("Функция absPlus не найдена.");

    binOp = null;

    // Найти поставщика для absMinus и получить функцию.
    for(BinFuncProvider bfp : ldr) {
      if(bfp.get().getName().equals("absMinus")) {
        binOp = bfp.get();
        break;
      }
    }

    if(binOp != null)
      System.out.println("Результат выполнения функции absMinus: " +
                         binOp.func(12, -4));
    else
      System.out.println("Функция absMinus не найдена.");

  }
}
```

Давайте внимательно посмотрим, каким образом служба загружается и выполняется в предыдущем коде. Сначала с помощью следующего оператора создается загрузчик служб для служб типа BinFuncProvider:

```
ServiceLoader<BinFuncProvider> ldr =
  ServiceLoader.load(BinFuncProvider.class);
```

Обратите внимание, что в качестве параметра типа для ServiceLoader указывается BinFuncProvider. Тот же самый тип используется и при вызове load(), т.е. поставщики, реализующие данный интерфейс, будут найдены. Таким образом, после выполнения этого оператора классы BinFuncProvider в модуле будут доступны через ldr. Здесь будут доступны как AbsPlusProvider, так и AbsMinusProvider.

Затем объявляется ссылка типа BinaryFunc по имени binOp, которая инициализируется значением null. Она будет применяться для ссылки на реализацию, предоставляющую конкретный тип бинарной функции. Далее в представленном ниже цикле в ldr ищется функция с именем "absPlus":

```
// Найти поставщика для absPlus и получить функцию.
for(BinFuncProvider bfp : ldr) {
  if(bfp.get().getName().equals("absPlus")) {
    binOp = bfp.get();
    break;
  }
}
```

В цикле for стиля "for-each" осуществляется проход по ldr с проверкой имени функции, предоставленной поставщиком. Если имя совпадает с "absPlus", тогда такая функция присваивается binOp путем вызова метода get() поставщика. Наконец, если функция найдена, как будет в рассматриваемом примере, то она выполняется с помощью следующего оператора:

```
if(binOp != null)
  System.out.println("Результат выполнения функции absPlus: " +
                     binOp.func(12, -4));
```

Поскольку binOp ссылается на экземпляр AbsPlus, вызов func() выполняет сложение абсолютных значений. Аналогичная последовательность используется для поиска и выполнения функции AbsMinus.

Так как в MyModAppDemo теперь применяется BinFuncProvider, файл определения его модуля должен включать оператор uses, указывающий на данный факт. Вспомните, что класс MyModAppDemo находится в модуле appstart, а потому понадобится изменить содержимое файла module-info.java для appstart:

```
// Определение модуля для главного приложения.
// Он теперь использует BinFuncProvider.
module appstart {
  // Затребовать модули appfuncs и userfuncs.
  requires appfuncs;
  requires userfuncs;
  // appstart теперь использует BinFuncProvider.
  uses userfuncs.binaryfuncs.BinFuncProvider;
}
```

Компиляция и запуск модульного приложения со службами

Выполнив все предыдущие шаги, можете скомпилировать и запустить пример с помощью следующих команд:

```
javac -d appmodules --module-source-path appsrc
  appsrc\userfuncsimp\module-info.java
  appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java

java --module-path appmodules -m appstart/appstart.mymodappdemo.MyModAppDemo
```

Вот вывод:

```
2 является делителем 10
Наименьший общий делитель для 35 и 105 равен 5
Наибольший общий делитель для 35 и 105 равен 7
Результат выполнения функции absPlus: 16
Результат выполнения функции absMinus: 8
```

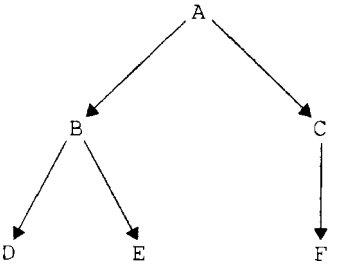
В выводе видно, что бинарные функции были обнаружены и выполнены. Важно подчеркнуть, что в случае отсутствия либо оператора `provides` в модуле `userfuncsimp`, либо оператора `uses` в модуле `appstart` приложение работать не будет.

И последний момент. Предыдущий пример был очень простым, чтобы ясно проиллюстрировать поддержку служб со стороны модулей, но возможны и более сложные применения. Например, службу можно было бы использовать для предоставления метода `sort()`, который сортирует файл, и предусмотреть разнообразные алгоритмы сортировки, доступные через службу. Тогда конкретный алгоритм сортировки выбирался бы на базе желаемых характеристик времени выполнения, природы и/или размера данных, а также наличия поддержки произвольного доступа к данным. Возможно, вы попробуете реализовать такую службу, экспериментируя со службами в модулях.

Графы модулей

При работе с модулями вполне вероятно вы столкнетесь с термином *графы модулей*. На этапе компиляции компилятор распознает отношения зависимостей между модулями, создавая граф модуля, который представляет зависимости. Такой процесс обеспечивает распознавание *всех зависимостей*, в том числе косвенных. Скажем, если модуль `A` требует модуль `B`, а `B` — модуль `C`, то граф модулей будет содержать модуль `C`, даже когда `A` не использует его напрямую.

Графы модулей могут быть изображены визуально в целях иллюстрации взаимосвязей между модулями. Рассмотрим простой пример. Пусть имеются шесть модулей с именами `A`, `B`, `C`, `D`, `E` и `F`. Предположим, что модуль `A` требует `B` и `C`, модуль `B` — `D` и `E`, а модуль `C` — `F`. Ниже такие взаимосвязи представлены визуально. (Поскольку модуль `java.base` включается автоматически, на диаграмме он не показан.)



В графе модулей Java стрелки направлены от зависимого модуля к модулю, который затребован. Таким образом, на изображенном выше графе модулей видно, у каких модулей есть доступ к другим модулям. Откровенно говоря, визуально представить графы модулей удается только в самых маленьких приложениях по причине сложности, обычно присущей многим коммерческим приложениям.

Три специальных характерных черты модулей

В предшествующих обсуждениях затрагивались ключевые характеристики модулей, поддерживаемых языком Java, и именно на них вы обычно полагаетесь при создании собственных модулей. Тем не менее, модули обладают еще тремя характерными чертами, которые могут оказаться крайне полезными в определенных обстоятельствах. Речь идет об открытых модулях, операторе `opens` и операторе `require static`. Каждая характеристика рассчитана на конкретную ситуацию и представляет собой довольно развитый аспект системы модулей. Однако всем программистам на Java важно иметь общее представление об их назначении.

Открытые модули

Ранее в главе вы узнали, что по умолчанию типы в пакетах модуля доступны только в том случае, если они явно экспортированы через оператор `exports`. Хотя обычно это нежелательно, могут сложиться обстоятельства, когда полезно разрешить доступ во время выполнения ко всем пакетам в модуле вне зависимости от того, экспортируется пакет или нет. В таком случае можно создать *открытый модуль*, который объявляется за счет помещения перед ключевым словом `module` модификатора `open`:

```
open module имя-модуля {
  // определение модуля
}
```

Во время выполнения открытый модуль разрешает доступ к типам из всех его пакетов. Тем не менее, важно понимать, что на этапе компиляции доступны только те пакеты, которые экспортируются явно. Таким образом, модификатор `open` влияет только на доступность во время выполнения. Основной причиной существования открытых модулей является обеспечение доступа

к пакетам в модуле через рефлексию. Как объяснялось в главе 12, рефлексия представляет собой средство, позволяющее программе анализировать код во время выполнения.

Оператор opens

В модуле можно открыть конкретный пакет для доступа во время выполнения со стороны других модулей и механизма рефлексии, а не открывать весь модуль. Для такой цели предназначен оператор opens:

```
opens packageName;
```

На месте packageName указывается имя пакета, к которому должен быть открыт доступ. Допускается также добавлять конструкцию to со списком модулей, для которых открывается пакет.

Важно понимать, что оператор opens не предоставляет доступ на этапе компиляции. Он применяется только для открытия доступа к пакету во время выполнения и со стороны рефлексии. Однако вы можете экспортировать и открывать модуль. И еще один аспект: оператор opens нельзя использовать в открытом модуле. Не забывайте, что все пакеты в открытом модуле уже открыты.

Оператор requires static

Как вам известно, оператор requires устанавливает зависимость, которая по умолчанию применяется как во время компиляции, так и во время выполнения. Тем не менее, такую зависимость можно ослабить, чтобы модуль не требовался во время выполнения. Цель достигается за счет использования в операторе requires модификатора static. Скажем, следующий оператор указывает, что модуль mymod требуется на этапе компиляции, но не во время выполнения:

```
requires static mymod;
```

В данном случае добавление static делает mymod необязательным во время выполнения, что полезно в ситуации, когда программа может задействовать функциональность, если она присутствует, но не требует ее.

Введение в jlink и файлы модулей JAR

Как обсуждалось ранее, модули представляют собой существенное усовершенствование языка Java. Система модулей также поддерживает расширения во время выполнения. Одной из наиболее важных является возможность создания образа времени выполнения, который специально приспособлен к вашему приложению. Для этого можно применять инструмент JDK под названием jlink, объединяющий группу модулей в оптимизированный образ времени выполнения. Инструмент jlink можно использовать для связывания модульных файлов JAR, файлов JMOD или даже модулей в их неархивированной форме с "развернутым каталогом".

Связывание файлов в развернутом каталоге

Давайте сначала ознакомимся с применением `jlink` для создания образа времени выполнения из неархивированных модулей, т.е. файлы содержатся внутри полностью развернутого каталога в необработанном виде. При работе в среде Windows показанная далее команда связывает модули для первого примера из этой главы. Команда должна вводиться в каталоге, расположенном *непосредственно над* `mymodapp`.

```
jlink --launcher MyModApp=appstart/appstart.mymodappdemo.MyModAppDemo
      --module-path "%JAVA_HOME%"\jmods;mymodapp\appmodules
      --add-modules appstart --output mylinkedmodapp
```

Проанализируем команду. Первый параметр, `--launcher`, сообщает инструменту `jlink` о необходимости создания команды, которая запускает приложение. В нем указываются имя приложения и путь к главному классу. В данном случае главным классом является `MyModAppDemo`. В параметре `--module-path` задаются пути к требующимся модулям: путь к модулям платформы и путь к модулям приложения. Обратите внимание на использование переменной среды `JAVA_HOME`, которая представляет путь к стандартному каталогу JDK. Например, в стандартной установке Windows такой путь обычно будет выглядеть как `"C:\program files"\java\jdk-17\jmods`, но за счет применения `JAVA_HOME` путь будет короче и сможет работать независимо от того, в каком каталоге был установлен комплект JDK. С помощью параметра `--add-modules` указывается добавляемый модуль (или модули). Здесь задан только модуль `appstart`, потому что `jlink` автоматически распознает все зависимости и включает все обязательные модули. Наконец, в параметре `--output` указан выходной каталог.

После выполнения предыдущей команды будет создан каталог по имени `mylinkedmodapp`, содержащий образ времени выполнения. В его подкаталоге `bin` вы обнаружите файл запуска `MyModApp`, который можно использовать для запуска приложения. Скажем, в Windows это будет командный файл, выполняющий программу.

Связывание модульных файлов JAR

Несмотря на удобство связывания модулей из развернутого каталога, при работе с реальным кодом чаще придется иметь дело с файлами JAR. (Вспомните, что JAR означает *J*ava *AR*chive (архив Java) и является файловым форматом, обычно применяемым при развертывании приложений.) В случае модульного кода будут использоваться *модульные файлы JAR*. Модульный файл JAR содержит файл `module-info.class`. Начиная с версии JDK 9, инструмент `jar` позволяет создавать модульные файлы JAR. Например, теперь он способен распознавать путь к модулю. После создания модульных файлов JAR вы можете с помощью `jlink` связать их в образ времени выполнения. Чтобы лучше понять процесс, давайте снова обратимся к первому примеру,

рассмотренному в текущей главе. Ниже приведены команды jar, которые создают модульные файлы JAR для программы MyModAppDemo. Каждая команда должна вводиться в каталоге, расположенном непосредственно над mymodapp. Вдобавок понадобится создать внутри mymodapp подкаталог по имени applib.

```
jar --create --file=mymodapp\applib\appfuncs.jar
    -C mymodapp\appmodules\appfuncs .
jar --create --file=mymodapp\applib\appstart.jar
    --main-class=appstart.mymodappdemo.MyModAppDemo
    -C mymodapp\appmodules\appstart .
```

Здесь параметр --create сообщает jar о необходимости создания нового файла JAR. В параметре --file указывается имя файла JAR. Включаемые файлы задаются в параметре -C. Класс, содержащий метод main(), указывается в параметре --main-class. После выполнения этих команд файлы JAR для приложения будут находиться в подкаталоге applib внутри mymodapp.

Имея только что созданные модульные файлы JAR, можно ввести команду, которая их свяжет:

```
jlink --launcher MyModApp=appstart
      --module-path "%JAVA_HOME%"\jmods;mymodapp\applib
      --add-modules appstart --output mylinkedmodapp
```

В данном случае в параметре --module-path указывается путь к файлам JAR, а не путь к развернутым каталогам. В остальном команда jlink такая же, как и прежде.

Интересно отметить, что с применением следующей команды можно запускать приложение прямо из файлов JAR. Она должна вводиться в каталоге, расположенном непосредственно над mymodapp.

```
java -p mymodapp\applib -m appstart
```

В параметре -p указан путь к модулю, а в -m — модуль, содержащий точку входа программы.

Файлы JMOD

Инструмент jlink также способен связывать файлы в более новом формате JMOD, появившемся в версии JDK 9. Файлы JMOD могут включать элементы, которые неприменимы к файлу JAR, и создаются посредством нового инструмента jmod. Хотя в большинстве приложений по-прежнему будут использоваться модульные файлы JAR, файлы JMOD окажутся полезными в специализированных ситуациях. Интересен и тот факт, что начиная с JDK 9, модули платформы содержатся в файлах JMOD.

На заметку! Инструмент jlink также может быть задействован недавно добавленным инструментом jpackage, который позволяет создавать собственное устанавливаемое приложение.

Кратко об уровнях и автоматических модулях

При изучении модулей, вероятно, вы встретите ссылки на два дополнительных средства, относящиеся к модулям: *уровни* и *автоматические модули*. Оба средства предназначены для специализированной расширенной работы с модулями или во время проведения миграции существующих приложений. Вполне вероятно, что большинству программистов эти средства применять не придется, но ради полноты картины ниже даны их краткие описания.

Уровень модулей связывает модули в графе модулей с загрузчиком классов. Таким образом, на разных уровнях могут использоваться разные загрузчики классов. Уровни упрощают создание определенных специализированных типов приложений.

Автоматический модуль создается за счет указания немодульного файла JAR в пути к модулям, а его имя генерируется автоматически. (Также можно явно указать имя автоматического модуля в файле манифеста.) Автоматические модули позволяют нормальным модулям иметь зависимость от кода из автоматических модулей. Автоматические модули предоставляются для помощи в переносе кода, написанного до появления модулей, в полностью модульный код. Следовательно, они являются главным образом средством перехода.

Заключительные соображения по поводу модулей

В предшествующих обсуждениях были описаны и продемонстрированы основные элементы системы модулей Java, о которых каждый программист на Java должен иметь хотя бы базовое представление. Как вы могли догадаться, система модулей предлагает дополнительные средства, которые обеспечивают детальный контроль над созданием и использованием модулей. Например, инструменты `javac` и `java` поддерживают гораздо больше параметров, связанных с модулями, чем было упомянуто в главе. Поскольку модули являются важным дополнением к Java, вполне вероятно, что система модулей со временем будет развиваться, поэтому имеет смысл следить за усовершенствованиями данного инновационного аспекта Java.

ГЛАВА

17 Выражения switch, записи и прочие недавно добавленные средства

Характерной чертой языка Java была его способность адаптироваться к растущим требованиям современной вычислительной среды. В течение прошедших лет в Java появилось много новых функциональных средств, каждое из которых реагировало на изменения в вычислительной среде либо на инновации в проектировании языков программирования. Такой непрерывный процесс позволил Java оставаться одним из самых важных и популярных языков программирования в мире. Как объяснялось ранее, книга была обновлена с учетом JDK 17 — версии Java с долгосрочной поддержкой (LTS). В состав JDK 17 входит несколько новых языковых средств, которые были добавлены в Java с момента выхода предыдущей версии LTS, т.е. JDK 11. Ряд небольших добавлений были описаны в предшествующих главах. В настоящей главе рассматриваются избранные основные дополнения, в том числе:

- расширения оператора `switch`;
- текстовые блоки;
- записи;
- шаблоны в `instanceof`;
- запечатанные классы и интерфейсы.

Далее приводятся краткие описания каждого дополнения. Оператор `switch` был расширен несколькими способами, наиболее важным из которых является *выражение* `switch`. Выражение `switch` наделяет `switch` способностью выпускать значение. *Текстовые блоки* позволяют строковому литералу занимать более одной строчки. Записи, поддерживаемые новым ключевым словом `record`, дают возможность создавать класс, специально предназначенный для хранения группы значений. Кроме того, была добавлена вторая форма операции `instanceof`, использующая шаблон типа. С ее помощью можно указывать переменную, которая получает экземпляр проверяемого типа, если `instanceof` завершается успешно. Теперь можно определять *запечатанный* класс или интерфейс. Запечатанный класс может наследоваться только явно заданными подклассами. Запечатанный интерфейс может быть реализован только явно указанными классами или расширен только явно заданными интерфейсами. Таким образом, запечатывание класса или интерфейса обеспечивает точный контроль над его наследованием и реализацией.

Расширения оператора switch

Оператор switch с самого начала входил в состав языка Java. Он представляет собой важнейший элемент управляющих операторов программы на Java и обеспечивает множественное ветвление. Более того, оператор switch настолько существенен в программировании, что ту или иную его форму можно обнаружить в других популярных языках программирования. Традиционная форма switch, которая всегда была частью Java, рассматривалась в главе 5. Начиная с версии JDK 14, оператор switch был существенно расширен за счет добавления четырех новых функциональных возможностей:

- выражение switch;
- оператор yield;
- поддержка списка констант case;
- оператор case со стрелкой.

Все новые функциональные возможности будут подробно обсуждаться далее в главе, но вот их краткое описание. Выражение switch — это по существу оператор switch, который производит значение. Таким образом, выражение switch можно применять, например, в правой части операции присваивания. Значение, созданное выражением switch, указывается с помощью оператора yield. Теперь можно иметь более одной константы в операторе case за счет использования списка констант. Добавлена вторая форма оператора case, в которой вместо двоеточия применяется стрелка (->), предоставляющая ему новые возможности.

В совокупности расширения оператора switch являются довольно значительными изменениями языка Java. Они не только предоставляют новые возможности, но в некоторых ситуациях также предлагают превосходные альтернативы традиционным подходам. Из-за этого важно четко понимать, что, как и почему стоит за новыми функциями switch.

Один из лучших способов понять расширения оператора switch — начать с примера, в котором используется традиционный оператор switch, а затем постепенно добавлять каждое новое средство, что позволит сделать очевидными применение и пользу от расширений. Для начала вообразите себе, что какое-то устройство генерирует целочисленные коды, обозначающие различные события, и с каждым кодом события вы хотите ассоциировать уровень приоритета. Большинство событий будут иметь обычный приоритет, но некоторые получат более высокий приоритет. Вот программа, в которой используется традиционный оператор switch для предоставления уровня приоритета с учетом кода события:

```
// Использование традиционного оператора switch для установки
// уровня приоритета на основе кода события.
class TraditionalSwitch {
  public static void main(String[] args) {
    int priorityLevel;
    int eventCode = 6010;
```

```
    // Традиционный оператор switch, который предоставляет
    // значение, ассоциированное с case.
    switch(eventCode) {
      case 1000:     // в традиционном операторе switch используется
                     // укладка операторов case
      case 1205:
      case 8900:
        priorityLevel = 1;
        break;
      case 2000:
      case 6010:
      case 9128:
        priorityLevel = 2;
        break;
      case 1002:
      case 7023:
      case 9300:
        priorityLevel = 3;
        break;
      default:   // нормальный приоритет
        priorityLevel = 0;
    }
    System.out.println("Уровень приоритета для кода события "
                       + eventCode + " равен " + priorityLevel);
  }
}
```

Вот вывод, генерируемый программой:

```
Уровень приоритета для кода события 6010 равен 2
```

Разумеется, с применением традиционного оператора `switch`, как было показано в программе, не связано ничего плохого и именно так код Java пишется уже более двух десятилетий. Однако в последующих разделах вы увидите, что во многих случаях традиционный оператор `switch` можно улучшить за счет использования его расширенных возможностей.

Использование списка констант case

Мы начнем с одного из простейших способов модернизации традиционного оператора `switch`: применения списка констант `case`. В прошлом, когда две константы обрабатывались одной и той же кодовой последовательностью, использовалась *укладка операторов* `case`, и именно такой прием применялся в предыдущей программе. Например, вот как выглядят операторы `case` для значений 1000, 1205 и 8900:

```
case 1000:     // в традиционном операторе switch используется
               // укладка операторов case
case 1205:
case 8900:
  priorityLevel = 1;
  break;
```

Укладка операторов case позволяет всем трем конструкциям case использовать одну и ту же кодовую последовательность, предназначенную для установки переменной priorityLevel в 1. Как объяснялось в главе 5, в switch традиционного стиля укладка операторов case становится возможной из-за того, что поток выполнения проходит через все case, пока не встретится break. Хотя такой подход работает, более элегантное решение предусматривает применение *списка констант* case.

Начиная с JDK 14, в одном операторе case можно указывать более одной константы case, просто разделяя их запятыми. Например, ниже показан более компактный вид оператора case для значений 1000, 1205 и 8900:

```
case 1000, 1205, 8900:        // использовать список констант case
  priorityLevel = 1;
  break;
```

Перепишем полный оператор switch для использования списка констант case:

```
// В этом операторе switch применяется список констант case.
switch(eventCode) {
  case 1000, 1205, 8900:
    priorityLevel = 1;
    break;
  case 2000, 6010, 9128:
    priorityLevel = 2;
    break;
  case 1002, 7023, 9300:
    priorityLevel = 3;
    break;
  default:                    // нормальный приоритет
    priorityLevel = 0;
}
```

Как видите, количество операторов case сократилось до трех, улучшив читабельность и управляемость оператора switch. Несмотря на то что поддержка списка констант case сама по себе не добавляет в switch какие-либо фундаментально новые возможности, она помогает оптимизировать код. Во многих ситуациях поддержка списка констант case также предлагает простой способ улучшения существующего кода, особенно если ранее широко применялась укладка операторов case. Таким образом, она является средством, которое можно запустить в работу немедленно при минимальном переписывании кода.

Появление выражения switch и оператора yield

Среди расширений оператора switch наибольшее влияние окажет *выражение* switch, которое представляет собой по существу оператор switch, возвращающий значение. Таким образом, оно обладает всеми возможностями традиционного оператора switch, а также способностью производить результат, что делает выражение switch одним из самых важных дополнений языка Java за последние годы.

Предоставить значение выражения switch можно с помощью нового оператора yield, который имеет следующий общий вид:

```
yield value;
```

Здесь значение value производится оператором switch и может быть любым выражением, совместимым с требуемым типом значения. Ключевой для понимания аспект, связанный с оператором yield, состоит в том, что он немедленно завершает switch. Следовательно, он работает примерно так же, как break, но обладает дополнительной возможностью предоставлять значение. Важно отметить, что yield — контекстно-чувствительное ключевое слово, т.е. yield за рамками выражения switch является просто идентификатором без специального смысла. Тем не менее, в случае использования метода по имени yield() он должен быть уточнен. Например, если внутри класса определен нестатический метод yield(), тогда придется вызывать его как this.yield().

Указать выражение switch очень легко. Необходимо просто разместить switch в контексте, где требуется значение, скажем, в правой части оператора присваивания, в аргументе метода или в возвращаемом из метода значении. В приведенной далее строке задействовано выражение switch:

```
int x = switch(y) {  // ...
```

Здесь результат switch присваивается переменной x. С применением выражения switch связан один важный момент: каждый оператор case (плюс default) должен производить значение (если только не генерируется исключение). Другими словами, каждый путь через выражение switch должен обеспечивать результат.

Добавление выражения switch упрощает написание кода в ситуациях, когда каждый оператор case устанавливает значение некоторой переменной. Такие ситуации могут возникать по-разному. Скажем, каждый case может устанавливать булевскую переменную, которая указывает на успех или неудачу какого-либо действия, предпринятого в switch. Однако зачастую установка переменной оказывается *основной целью* switch, как в случае оператора switch, используемого в предыдущей программе. Его работа была связана с установлением уровня приоритета, ассоциированного с кодом события. В традиционном операторе switch каждый оператор case должен индивидуально присваивать значение переменной, и эта переменная становится де-факто результатом переключения. Именно такой подход применялся в предшествующих программах, где каждый оператор case устанавливал значение переменной priorityLevel. Хотя этот подход десятилетиями использовался в программах на Java, выражение switch предлагает более эффективное решение, поскольку желаемое значение создается самим оператором switch.

Предыдущее обсуждение воплощено в следующей версии программы за счет замены оператора switch выражением switch:

```
// Преобразование оператора switch в выражение switch.
class SwitchExpr {

  public static void main(String[] args) {
    int eventCode = 6010;

    // Это выражение switch. Обратите внимание на то, как его значение
    // присваивается переменной priorityLevel, а также на то,
    // что значение switch предоставляется оператором yield.
    int priorityLevel = switch(eventCode) {
      case 1000, 1205, 8900:
        yield 1;
      case 2000, 6010, 9128:
        yield 2;
      case 1002, 7023, 9300:
        yield 3;
      default:  // нормальный приоритет
        yield 0;
    };

    System.out.println("Уровень приоритета для кода события "
                       + eventCode + " равен " + priorityLevel);
  }
}
```

Взгляните внимательно на оператор `switch` в программе. Обратите внимание, что он существенно отличается от того, который применялся в предшествующих примерах. Вместо присваивания `priorityLevel` значения внутри каждого оператора `case` в текущей версии программы переменной `priorityLevel` присваивается результат самого `switch`. Таким образом, требуется только одно присваивание `priorityLevel` и длина `switch` уменьшается. Использование выражения `switch` также гарантирует выдачу значения каждым оператором `case`, что не позволит забыть об указании значения для `priorityLevel` в каком-нибудь `case`. Кроме того, значение `switch` выпускается оператором `yield` внутри каждого оператора `case`. Как упоминалось ранее, `yield` вызывает немедленное прекращение `switch`, поэтому сквозной проход от `case` до `case` не произойдет. В итоге оператор `break` не требуется и не допускается. Еще один момент, который следует отметить, касается точки с запятой после закрывающей скобки `switch`. Поскольку данный `switch` участвует в операторе присваивания, он обязан заканчиваться точкой с запятой.

Существует важное ограничение, применяемое к выражению `switch`: операторы `case` должны охватывать все значения, которые могут возникать. Таким образом, выражение `switch` обязано быть *исчерпывающим*. Скажем, если управляющее выражение в `switch` имеет тип `int`, то должны обрабатываться все значения `int`, что, конечно же, породило бы огромное количество операторов `case`! По этой причине большинство выражений `switch` будут иметь оператор `default`. Исключением из указанного правила является ситуация, когда используется перечисление и каждому значению перечисления соответствует свой оператор `case`.

Появление стрелки в операторе case

Наряду с тем, что применение yield в предыдущей программе — совершенно допустимый способ указать значение для выражения switch, он не является единственным. Во многих ситуациях более простой способ предоставления значения предусматривает использование новой формы case, в которой двоеточие заменяется стрелкой (->). Например, следующую строку:

```
case 'X': // ...
```

можно переписать, задействовав стрелку:

```
case 'X' -> // ...
```

Во избежание путаницы в этом обсуждении форма оператора case, содержащая стрелку, будет называться case *со стрелкой*, а традиционная форма case, которая включает двоеточие — case *с двоеточием*. Хотя обе показанные выше формы будут соответствовать символу X, точное действие каждого стиля оператора case отличается в трех очень важных отношениях.

Во-первых, поток выполнения после case со стрелкой не переходит к следующему оператору case. Таким образом, нет никакой необходимости применять break. Выполнение просто завершается в конце case со стрелкой. Хотя природа сквозного выполнения традиционного оператора case с двоеточием всегда была присуща Java, она подвергалась критике, поскольку могла служить источником ошибок, скажем, если программист забыл добавить оператор break, чтобы предотвратить сквозное выполнение, когда оно нежелательно. Избежать такой ситуации позволяет case со стрелкой. Во-вторых, case со стрелкой обеспечивает "сокращенный" способ предоставления значения в случае использовании в выражении switch. Именно по этой причине case со стрелкой часто применяется в выражениях switch. В-третьих, целью case со стрелкой должно быть выражение, блок либо исключение. Ею не может быть последовательность операторов, как разрешено в традиционном операторе case. Таким образом, case со стрелкой будет иметь одну из представленных далее общих форм:

```
case константа -> выражение;
case константа -> { блок-операторов }
case константа -> throw ...
```

Разумеется, в основном используются первые две формы.

Вероятно, чаще всего case со стрелкой применяется в выражении switch и наиболее распространенной целью case со стрелкой является выражение, поэтому с такой ситуации мы и начнем. Когда в качестве цели case со стрелкой указано выражение, его значение становится значением выражения switch при совпадении с данным оператором case. Таким образом, во многих обстоятельствах он обеспечивает очень эффективную альтернативу оператору yield. Например, вот первый case в примере с приоритетом для кода события, переписанный для использования case со стрелкой:

```
case 1000, 1205, 8900 -> 1;
```

Здесь значение выражения (равное 1) автоматически становится значением, которое производит switch при соответствии с данным оператором case. Другими словами, выражение будет значением, выдаваемым switch. Обратите внимание, что оператор довольно компактен, но ясно выражает намерение предоставить значение.

В следующей программе все выражение switch полностью переписано для применения case со стрелкой:

```
// Использование "сокращенной" формы со стрелкой
// для предоставления уровня приоритета.
class SwitchExpr2 {

  public static void main(String[] args) {
    int eventCode = 6010;
    // Обратите внимание в этом выражении switch на то, как значение
    // предоставляется с применением case со стрелкой. Кроме того,
    // для предотвращения сквозного выполнения оператор break
    // не требуется (и не разрешен).
    int priorityLevel = switch(eventCode) {
      case 1000, 1205, 8900 -> 1;
      case 2000, 6010, 9128 -> 2;
      case 1002, 7023, 9300 -> 3;
      default -> 0;  // нормальный приоритет
    };
    System.out.println("Уровень приоритета для кода события "
                       + eventCode + " равен " + priorityLevel);
  }
}
```

Программа выдает тот же результат, что и ее предыдущая версия. Глядя на код, легко понять, почему форма case со стрелкой настолько подходит для многих видов выражений switch. Она компактна и устраняет необходимость в отдельном операторе yield. Поскольку case со стрелкой не допускает сквозного выполнения, то нет необходимости в операторе break. Каждый case заканчивается выдачей значения своего выражения. Кроме того, сравнив окончательную версию switch с традиционной версией switch, приведенной в начале обсуждения, вам станет совершенно ясно, насколько оптимизированной и выразительной является новая версия. Совместно расширения оператора switch предлагают действительно впечатляющий способ улучшения ясности и отказоустойчивости кода.

Подробный анализ оператора case со стрелкой

Оператор case со стрелкой обеспечивает значительную гибкость. Прежде всего, при использовании в форме выражения выражение может относиться к любому типу. Например, ниже показан допустимый оператор case:

```
case -1 -> getErrorCode();
```

Результат вызова getErrorCode() становится значением объемлющего выражения switch. Вот еще один пример:

```
case 0 -> normalCompletion = true;
```

В данном случае результат присваивания, равный true, становится выдаваемым значением. Ключевой момент здесь связан с тем, что для цели оператора case со стрелкой может применяться любое допустимое выражение, если оно совместимо с типом, затребованным оператором switch.

Как уже упоминалось, целью -> также может служить блок кода. Всякий раз, когда нужно более одного выражения, в качестве цели case со стрелкой должен использоваться блок. Например, все операторы case в следующей версии программы, обрабатывающей коды событий, устанавливают значение булевской переменной по имени stopNow, указывая на необходимость немедленного завершения. Кроме того, программа выдает уровень приоритета.

```
// Использование блоков в операторах case со стрелками.
class BlockArrowCase {

  public static void main(String[] args) {
    boolean stopNow;

    int eventCode = 9300;

    // Использовать блоки в операторах case со стрелками. Снова
    // обратите внимание, что для предотвращения сквозного выполнения
    // оператор break не нужен (да и не разрешен).
    // Поскольку целью -> является блок, для предоставления значения
    // должен применяться оператор yield.
    int priorityLevel = switch(eventCode) {
      case 1000, 1205, 8900 -> {  // использовать блок
        stopNow = false;
        System.out.println("Тревога");
        yield 1;
      }
      case 2000, 6010, 9128 -> {
        stopNow = false;
        System.out.println("Предупреждение");
        yield 2;
      }
      case 1002, 7023, 9300 -> {
        stopNow = true;
        System.out.println("Опасность");
        yield 3;
      }
      default -> {
        stopNow = false;
        System.out.println("Все нормально");
        yield 0;
      }
    };

    System.out.println("Уровень приоритета для кода события "
                       + eventCode + " равен " + priorityLevel);
    if(stopNow) System.out.println("Требуется останов.");
  }
}
```

Программа генерирует такой вывод:

```
Опасность
Уровень приоритета для кода события 9300 равен 3
Требуется останов.
```

Как было продемонстрировано в примере, в случае применения блока для поставки значения выражению switch должен использоваться оператор yield. Кроме того, несмотря на применение блочных целей, каждый путь через выражение switch по-прежнему обязан предоставлять значение.

Хотя предыдущая программа обеспечивает простую иллюстрацию блочной цели case со стрелкой, она также поднимает интересный вопрос. Обратите внимание, что каждый оператор case в switch устанавливает значения двух переменных: priorityLevel с выдаваемым значением и stopNow. Можно ли каким-то образом заставить выражение switch выдавать более одного значения? В прямом смысле ответ будет отрицательным, потому что выражение switch способно выпускать только одно значение. Тем не менее, можно инкапсулировать два и больше значений внутри класса и выдавать объект этого класса. Начиная с JDK 16, в Java предлагается особенно простой и эффективный способ решения проблемы: запись. Как будет показано далее в главе, запись объединяет два или большее количество значений в одну логическую единицу. Что касается рассматриваемого примера, то запись может содержать значения priorityLevel и stopNow и выдаваться выражением switch как единое целое. Таким образом, запись снабжает switch удобным способом выдачи более одного значения.

Несмотря на то что оператор case со стрелкой очень полезен в выражении switch, важно подчеркнуть, что его использование этим не ограничивается. Оператор case со стрелкой также можно применять в операторе switch, что позволяет получать операторы switch, в которых не может происходить сквозной проход через case. В такой ситуации оператор yield не требуется (не разрешен) и switch не выпускает какое-либо значение. По существу он работает во многом подобно традиционному оператору switch, но без сквозного прохода. Ниже представлен пример:

```
// Использование операторов case со стрелками в операторе switch.
class StatementSwitchWithArrows {

  public static void main(String[] args) {
    int up = 0;
    int down = 0;
    int left = 0;
    int right = 0;
    char direction = 'R';

    // Использовать операторы case со стрелками в операторе switch.
    // Обратите внимание, что значение не производится.
    switch(direction) {
      case 'L' -> {
                    System.out.println("Повернуть налево");
                    left++;
                  }
```

```
      case 'R' -> {
                    System.out.println("Повернуть направо");
                    right++;
                  }
      case 'U' -> {
                    System.out.println("Двигаться вверх");
                    up++;
                  }
      case 'D' -> {
                    System.out.println("Двигаться вниз");
                    down++;
                  }
    }
    System.out.println(right);
  }
}
```

В приведенной программе `switch` является оператором, а не выражением, и на то имеются две причины. Во-первых, никакого значения не производится. Во-вторых, `switch` не является исчерпывающим, поскольку в нем отсутствует оператор `default`. (Вспомните, что выражения `switch` обязаны быть исчерпывающими, но это не касается операторов `switch`.) Однако обратите внимание, что из-за отсутствия сквозного прохода в `case` со стрелками оператор `break` не требуется. Интересно отметить, что поскольку в каждом операторе `case` увеличивается значение другой переменной, трансформировать такой `switch` в выражение было бы невозможно. Какое значение оно производило бы? Во всех четырех операторах `case` увеличиваются разные переменные.

И последнее замечание: внутри одного `switch` нельзя смешивать операторы `case` со стрелками и традиционные `case` с двоеточиями. Потребуется выбрать или одно, или другое. Например, следующая кодовая последовательность недопустима:

```
// Этот код не скомпилируется! Нельзя смешивать операторы case
// с двоеточиями и операторы case со стрелками.
switch(direction) {
  case 'L' -> {
                System.out.println("Повернуть налево");
                left++;
              }
  case 'R' :  // Ошибка! Смешивать разные стили case не разрешено.
              System.out.println("Повернуть направо");
              right++;
              break;
  case 'U' -> {
                System.out.println("Двигаться вверх");
                up++;
              }
  case 'D' -> {
                System.out.println("Двигаться вниз");
                down++;
              }
}
```

Еще один пример выражения switch

В заключение предоставленного здесь обзора расширений оператора switch приводится еще один пример, в котором используется выражение switch для выяснения, является ли буква гласной в английском языке. В примере задействованы все новые возможности switch. Обратите особое внимание на обработку буквы "Y". В английском языке "Y" может быть как гласной, так и согласной. Программа позволяет указать, как необходимо интерпретировать "Y", в зависимости от установки переменной yIsVowel. Чтобы справиться с этим особым случаем, для цели -> применяется блок операторов.

```
// Использование выражения switch для выяснения, является ли символ
// гласной буквой в английском языке. Обратите внимание на применение
// блока в качестве цели оператора case со стрелкой для Y.
class Vowels {
  public static void main(String[] args) {

    // Если буква Y должна считаться главной, тогда эту переменную
    // необходимо установить в true.
    boolean yIsVowel = true;

    char ch = 'Y';

    boolean isVowel = switch(ch) {
      case 'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U' -> true;
      case 'y', 'Y' -> { if(yIsVowel) yield true; else yield false; }
      default -> false;
    };

    if(isVowel) System.out.println(ch + " является гласной буквой.");
  }
}
```

В качестве эксперимента попробуйте переписать эту программу с использованием традиционного оператора switch. Вы обнаружите, что получится гораздо более длинная и менее управляемая версия. Новые расширения switch часто обеспечивают превосходное решение.

Текстовые блоки

Начиная с версии JDK 15, в Java обеспечивается поддержка *текстовых блоков*. Текстовый блок — это литерал нового типа, состоящий из последовательности символов, которые могут занимать более одной строчки. Текстовый блок сокращает утомительную работу, с которой программисты часто сталкиваются при создании длинных строковых литералов, потому что в текстовом блоке можно указывать символы новой строки без необходимости в применении управляющей последовательности \n. Кроме того, символы табуляции и двойных кавычек тоже можно вводить напрямую, не используя управляющую последовательность, и вдобавок есть возможность сохранить отступы в многострочной строке. Хотя поначалу текстовые блоки могут показаться относительно небольшим дополнением Java, они вполне могут стать одним из самых популярных средств.

Основы текстовых блоков

Текстовый блок поддерживается новым разделителем, который представляет собой три символа двойных кавычек: """. Текстовый блок создается помещением строки внутрь набора этих разделителей. В частности, текстовый блок начинается немедленно за символом новой строки после открывающего разделителя """. Таким образом, строчка, содержащая открывающий разделитель, должна заканчиваться символом новой строки. Текстовый блок начинается со следующей строчки. Текстовый блок заканчивается на первом символе закрывающего разделителя """. Вот простой пример:

```
"""
Текстовые блоки облегчают размещение последовательности
символов в нескольких строчках.
"""
```

В этом примере создается строка, в которой строчка "Текстовые блоки облегчают размещение последовательности" отделена от строчки "символов в нескольких строчках." символом новой строки. Для перехода на новую строку нет необходимости указывать управляющую последовательность \n. Таким образом, текстовый блок автоматически предохраняет символы новой строки в тексте. Как уже упоминалось, текстовый блок начинается с новой строчки, *следующей* за открывающим разделителем, и заканчивается в начале закрывающего разделителя, так что символ новой строки после второй строчки тоже сохраняется.

Важно подчеркнуть, что хотя в текстовом блоке применяется разделитель """, он относится к типу `String`. Таким образом, предыдущий текстовый блок можно присвоить строковой переменной:

```
String str = """
Текстовые блоки облегчают размещение последовательности
символов в нескольких строчках.
"""
```

В случае вывода `str` с использованием следующего оператора:

```
System.out.println(str);
```

вот что отобразится:

```
Текстовые блоки облегчают размещение последовательности
символов в нескольких строчках.
```

В приведенном примере обратите внимание еще на кое-что. Поскольку последняя строчка заканчивается символом новой строки, новая строка также будет присутствовать в результирующей строке. Если вам не нужен завершающий символ новой строки, тогда поместите закрывающий разделитель в конец последней строки, например:

```
String str = """
Текстовые блоки облегчают размещение последовательности
символов в нескольких строчках.""";   // теперь в конце символа
                                      // новой строки нет
```

Ведущие пробельные символы

В предыдущем примере текст в блоке был выровнен по левому краю, но поступать так не обязательно. В текстовом блоке могут присутствовать ведущие пробельные символы. Есть две основные причины, по которым могут понадобиться ведущие пробельные символы. Во-первых, они позволят лучше выровнять текст относительно уровня отступа кода вокруг него. Во-вторых, они поддерживают один или несколько уровней отступов внутри самого текстового блока.

Как правило, ведущие пробельные символы в текстовом блоке автоматически удаляются. Тем не менее, количество ведущих пробельных символов, подлежащих удалению из каждой строки, определяется числом ведущих пробельных символов в строке с наименьшим отступом. Например, если все строчки выровнены по левому краю, то никакие пробельные символы не удаляются. Если все строчки имеют отступ в два пробела, то из каждой строчки удаляются два пробела. Однако если одна строчка имеет отступ два пробела, вторая — четыре пробела и третья — шесть пробелов, тогда в начале каждой строчки удаляются только два пробела. В итоге удаляются нежелательные ведущие пробельные символы, но предохраняется отступ текста внутри блока. Описанный механизм проиллюстрирован в следующей программе:

```
// Демонстрация использования отступов в текстовом блоке.
class TextBlockDemo {

  public static void main(String[] args) {
    String str = """
                 Текстовые блоки поддерживают строки, которые
                    распространяются на две и большее количество
                       строчек и предохраняют отступы. Они позволяют
                    менее утомительно вводить длинные или сложные
                 строки в программе.
                 """;

    System.out.println(str);
  }
}
```

Вот вывод, генерируемый программой:

```
Текстовые блоки поддерживают строки, которые
   распространяются на две и большее количество
      строчек и предохраняют отступы. Они позволяют
   менее утомительно вводить длинные или сложные
строки в программе.
```

Как видите, ведущие пробельные символы были удалены до уровня крайних левых строчек, но не дальше. Таким образом, для текстового блока в программе может быть предусмотрен отступ, чтобы он лучше соответствовал уровню отступа кода, но с удалением ведущих пробельных символов, когда строка создается. Тем не менее, все пробельные символы после уровня отступа блока предохраняются.

Еще один ключевой момент: закрывающий разделитель """ участвует в определении объема удаляемых пробельных символов, поскольку он тоже может устанавливать уровень отступа. Следовательно, если закрывающий разделитель выровнен по левому краю, то пробельные символы не удаляются. В противном случае пробельные символы удаляются, пока не встретится первый символ текста или закрывающий разделитель. Например, рассмотрим показанную далее последовательность:

```
String str = """
    A
       B
    C
""";           // определяет начало отступа
String str2 = """
    A
       B
    C
      """;     // никакого влияния не оказывает
String str3 = """
    A
       B
    C
   """;        // приводит у удалению пробельных символов вплоть до """
System.out.print(str);
System.out.print(str2);
System.out.print(str3);
```

Вот вывод:

```
    A
       B
    C
A
   B
C
  A
     B
  C
```

Обратите особое внимание на размещение закрывающего разделителя для строки `str2`. Из-за того, что количество предшествующих пробелов в строчках, содержащих `A` и `C`, меньше числа пробелов перед """, закрывающий разделитель никак не влияет на то, сколько пробелов будет удалено.

Использование двойных кавычек в текстовом блоке

Другим важным преимуществом текстовых блоков является возможность указания двойных кавычек без необходимости в применении управляющей последовательности `\"`. Двойные кавычки в текстовом блоке трактуются подобно любому другому символу. Возьмем в качестве примера такую программу:

```
// Использование двойных кавычек в текстовом блоке.
class TextBlockDemo2 {
  public static void main(String[] args) {
    String str = """
     Внутри текстового блока можно указывать двойные кавычки без необходимости
     в использовании управляющих последовательностей. Например:
     Он сказал: "На крыше кот".
     Она спросила: "Как он туда попал?"
                  """;
    System.out.println(str);
  }
}
```

Вывод программы выглядит следующим образом:

```
Внутри текстового блока можно указывать двойные кавычки без необходимости
в использовании управляющих последовательностей. Например:
Он сказал: "На крыше кот".
Она спросила: "Как он туда попал?"
```

Как видите, использовать управляющую последовательность \" не пришлось. Кроме того, поскольку двойные кавычки рассматриваются как "нормальные" символы, нет необходимости применять их сбалансированным образом внутри текстового блока. Например, вот вполне приемлемый текстовый блок:

```
"""
""абв"
"""
```

Просто помните о том, что три двойные кавычки как единое целое определяют разделитель текстового блока.

Управляющие последовательности в текстовых блоках

В текстовом блоке разрешено использовать управляющие последовательности вроде \n или \t. Однако из-за того, что символы двойной кавычки, новой строки и табуляции можно вводить напрямую, управляющие последовательности будут требоваться нечасто. Тем не менее, вместе с текстовыми блоками в Java были добавлены две новые управляющие последовательности. Первой является \s, которая указывает пробел. Таким образом, ее можно применять для обозначения завершающих пробелов. Вторая управляющая последовательность — \конец-строки. Поскольку за символом \ должен следовать разделитель строки, он должен использоваться только в конце строчки. В текстовом блоке символ \ предотвращает включение символа новой строки в конец его строчки. Следовательно, символ \ по существу представляет собой индикатор продолжения строки. Например:

```
String str = """
             один \
             два
             три \
             четыре
             """;
System.out.println(str);
```

С учетом того, что символ новой строки после строк "один" и "три" подавляется, вывод будет таким:

```
один два
три четыре
```

Важно отметить, что управляющую последовательность `\конец-строки` можно применять только в текстовом блоке. Например, ее нельзя использовать для продолжения традиционного строкового литерала.

И последнее замечание, прежде чем закончить тему текстовых блоков: поскольку текстовые блоки обеспечивают более совершенный и простой способ ввода многих видов строк в программе, то ожидается, что они будут широко использоваться сообществом программистов на Java. Вполне вероятно, что вы начнете встречать их в коде, с которым работаете, или применять их в коде, который пишете сами. Просто помните, что версией JDK должна быть 15 или более поздняя.

Записи

Начиная с версии JDK 16, в Java поддерживается специальный класс, который называется *записью*. Запись разработана с целью обеспечения эффективного и простого в использовании способа для хранения группы значений. Скажем, вы можете применять запись для хранения набора координат, номеров и балансов банковских счетов, длины, ширины и высоты грузового контейнера и т.д. Поскольку запись содержит группу значений, на нее обычно ссылаются как на *агрегированный* тип. Однако запись — нечто большее, чем просто средство группирования данных, т.к. записи поддерживают ряд возможностей, присущих классам. Кроме того, запись обладает уникальными характеристиками, которые облегчают ее объявление и упрощают доступ к ее значениям. В результате записи позволяют гораздо легче работать с группами данных. Для записей предусмотрено новое контекстно-чувствительное ключевое слово `record`.

Одна из основных побудительных причин появления записей связана с желанием сократить усилия, необходимые для создания класса, основной целью которого была организация двух и более значений в одиночный блок. Хотя для этой цели всегда можно было использовать класс, такое решение влечет за собой написание немалого количества строк кода для конструкторов, методов получения и возможно переопределения одного или нескольких методов, унаследованных от `Object`. Как вы увидите, при создании агрегированного типа данных с применением записи упомянутые элементы обрабатываются автоматически, что значительно упрощает код. Еще одна причина добавления записей — позволить программе четко указывать, что предполагаемой целью класса является хранение группы данных, а не функционирование в качестве полнофункционального класса. Благодаря таким преимуществам записи оказались долгожданным дополнением языка Java.

Основы записей

Ранее уже упоминалось, что запись — это узконаправленный, специализированный класс. Она объявляется с помощью контекстно-чувствительного ключевого слова `record`, которое трактуется как ключевое слово только в контексте объявления записи. В других ситуациях `record` считается пользовательским идентификатором без особой значимости. Таким образом, добавление `record` не влияет на существующий код и не нарушает его работу.

Ниже показана общая форма объявления записи:

```
record имя-записи(список-компонентов) {
  // необязательные операторы тела
}
```

В приведенной общей форме видно, что объявление записи имеет существенные отличия от объявления класса. Во-первых, обратите внимание, что сразу после имени записи следует разделенный запятыми список объявлений параметров, называемый *списком компонентов*. В списке компонентов определяются данные, которые будут храниться в записи. Во-вторых, тело записи является необязательным благодаря тому, что компилятор автоматически предоставит элементы, необходимые для хранения данных, создаст конструктор записи и методы получения, которые обеспечат доступ к данным, а также переопределит методы `toString()`, `equals()` и `hashCode()`, унаследованные от `Object`. В результате во многих сценариях использования тело не требуется, т.к. само объявление `record` полностью определяет запись.

Вот пример простого объявления записи:

```
record Employee(String name, int idNum) {}
```

Запись имеет имя `Employee` и состоит из двух компонентов: строкового `name` и целочисленного `idNum`. В теле записи операторы не указаны и потому оно пустое. Такая запись может применяться для хранения имени и идентификационного номера сотрудника.

Учитывая только что показанное объявление `Employee`, определенные элементы создаются автоматически. Во-первых, для `name` и `idNum` объявляются закрытые финальные поля с типами `String` и `int`. Во-вторых, предоставляются открытые методы доступа, допускающие только чтение (методы получения), которые имеют такие же имена и типы, как у компонентов записи `name` и `idNum`, т.е. `name()` и `idNum()`. В общем случае для каждого компонента записи компилятор будет автоматически создавать соответствующее закрытое финальное поле и открытый метод получения, допускающий только чтение.

Еще одним элементом, который автоматически создается компилятором, будет *канонический конструктор* записи. Такой конструктор имеет список параметров, который содержит те же элементы в том же порядке, что и список компонентов в объявлении записи. Значения, передаваемые конструктору, автоматически присваиваются соответствующим полям в записи. Канонический конструктор в записи заменяет собой стандартный конструктор в классе.

Запись создается с помощью операции new тем же способом, каким создается экземпляр класса. Например, ниже создается новая запись Employee для имени "Doe, John" и идентификационного номера 1047:

```
Employee emp = new Employee("Doe, John", 1047);
```

После выполнения этого оператора закрытые поля name и idNum в emp будут содержать значения "Doe, John" и 1047. Таким образом, следующий оператор можно использовать для отображения информации, ассоциированной с emp:

```
System.out.println("Идентификатором сотрудника " + emp.name()
                   + " является " + emp.idNum());
```

Вот результирующий вывод:

```
Идентификатором сотрудника Doe, John является 1047
```

С записью связан один ключевой аспект: ее данные хранятся в закрытых финальных полях и предоставляются только методы получения. Следовательно, содержащиеся в записи данные являются неизменяемыми. Другими словами, после конструирования записи ее содержимое изменять нельзя. Тем не менее, если запись содержит ссылку на какой-то объект, тогда в этот объект можно вносить изменения, но не разрешено изменять то, на что ссылается ссылка в записи. Таким образом, в терминах Java говорят, что записи *поверхностно неизменяемы*.

Приведенное выше обсуждение воплощается в показанной далее программе. Записи часто применяются в качестве элементов списка. Скажем, компания может вести список записей Employee, предназначенных для хранения имен сотрудников вместе с соответствующими идентификационными номерами. Ниже демонстрируется простой пример такого использования. Здесь создается небольшой массив записей Employee и осуществляется проход в цикле по массиву с отображением содержимого каждой записи.

```
// Простой пример работы с записями.

// Объявить запись для сотрудника, что приведет к автоматическому
// созданию класса с закрытыми финальными полями, имеющими имена
// name и idNum, а также с методами доступа name() и idNum().
record Employee(String name, int idNum) {}

class RecordDemo {
  public static void main(String[] args) {
    // Создать массив записей Employee.
    Employee[] empList = new Employee[4];

    // Создать список сотрудников с использованием записи Employee.
    // Обратите внимание на способ конструирования каждой записи.
    // Аргументы автоматически присваиваются полям name и idNum
    // в создаваемой записи.
    empList[0] = new Employee("Doe, John", 1047);
    empList[1] = new Employee("Jones, Robert", 1048);
    empList[2] = new Employee("Smith, Rachel", 1049);
    empList[3] = new Employee("Martin, Dave", 1050);
```

```
    // Использовать методы доступа к содержимому записи
    // для отображения имен и идентификационных номеров.
    for(Employee e: empList)
      System.out.println("Идентификатором сотрудника " + e.name()
                         + " является " + e.idNum());
  }
}
```

Программа генерирует следующий вывод:

```
Идентификатором сотрудника Doe, John является 1047
Идентификатором сотрудника Jones, Robert является 1048
Идентификатором сотрудника Smith, Rachel является 1049
Идентификатором сотрудника Martin, Dave является 1050
```

До того как продолжить, важно упомянуть о нескольких ключевых моментах, касающихся записей. Прежде всего, запись не может быть унаследована от другого класса. Однако все записи неявно унаследованы от класса `java.lang.Record`, в котором переопределяются абстрактные методы `equals()`, `hashCode()` и `toString()`, объявленные в `Object`. Неявные реализации этих методов создаются автоматически на основе объявления записи. Тип записи не может быть расширен. Таким образом, все объявления записей считаются финальными. Хотя запись не может расширять другой класс, она может реализовывать один или несколько интерфейсов. За исключением `equals` применять имена методов, определенных в `Object`, для имен компонентов записи нельзя. Помимо полей, ассоциированных с компонентами записи, любые другие поля должны быть статическими. Наконец, запись может быть обобщенной.

Создание конструкторов записи

Хотя вы часто будете обнаруживать, что автоматически предоставляемый канонический конструктор — именно то, что требуется, можно также объявить один или несколько собственных конструкторов. Кроме того, можно определить собственную реализацию канонического конструктора. Желание объявить конструктор записи может возникать по ряду причин. Например, конструктор может проверять, входит ли значение в требуемый диапазон, выяснять, находится ли значение в корректном формате, удостоверяться в том, что объект реализует необязательную функциональность, или подтверждать, что аргумент не равен `null`. Для записи существуют два основных вида конструкторов, создаваемые явно: канонический и неканонический, и между ними есть некоторые различия. Ниже рассматривается создание каждого вида, начиная с определения собственной реализации канонического конструктора.

Объявление канонического конструктора

Несмотря на то что канонический конструктор имеет специфическую предопределенную форму, существуют два способа, которыми можно записать собственную реализацию. Во-первых, есть возможность явно объявить

полную форму канонического конструктора. Во-вторых, можно использовать так называемый *компактный конструктор записи*. Все подходы исследуются далее, начиная с полной формы.

Чтобы определить собственную реализацию канонического конструктора, просто поступайте так же, как с любым другим конструктором, указав имя записи и список ее параметров. Важно подчеркнуть, что для канонического конструктора типы и имена параметров обязаны совпадать с указанными в объявлении записи. Причина в том, что имена параметров связаны с автоматически создаваемыми полями и методами доступа, определенными в объявлении записи. Таким образом, они должны соответствовать как по типу, так и по имени. Кроме того, после завершения конструктора каждый компонент должен быть полностью инициализирован. Вдобавок применяются следующие ограничения: конструктор должен быть, по меньшей мере, таким же доступным, как в объявлении записи. Другими словами, если в записи указан модификатор доступа `public`, то конструктор тоже должен быть определен как открытый. Конструктор не может быть обобщенным и не может включать конструкцию `throws`. Он также не может вызывать другой конструктор, определенный для записи.

Вот пример записи `Employee`, в которой явно определен канонический конструктор, который используется для удаления любых ведущих или завершающих пробельных символов из имени, что гарантирует хранение имен в согласованной манере:

```
record Employee(String name, int idNum) {

  // Использование канонического конструктора для удаления любых ведущих
  // и завершающих пробелов, которые могут присутствовать в строке имени,
  // что гарантирует хранение имен в согласованной манере.
  public Employee(String name, int idNum) {
    // Удалить любые ведущие и завершающие пробелы.
    this.name = name.trim();
    this.idNum = idNum;
  }
}
```

Ведущие и/или завершающие пробельные символы удаляются в конструкторе вызовом `trim()`. Метод `trim()`, определенный в классе `String`, удаляет все ведущие и завершающие пробельные символы из строки и возвращает результат. (Если ведущие или завершающие пробельные символы отсутствуют, тогда исходная строка возвращается без изменений.) Результирующая строка присваивается полю `this.name`. В итоге поле `name` записи `Employee` никогда не будет содержать ведущие или завершающие пробелы. Затем значение `idNum` присваивается полю `this.idNum`. Поскольку идентификаторы `name` и `idNum` совпадают с именами полей, соответствующих компонентам `Employee`, а также с именами параметров канонического конструктора, имена полей должны уточняться посредством `this`.

Хотя, конечно же, нет ничего плохого в создании канонического конструктора, как только что было показано, часто можно применить более простой

способ, предусматривающий использование *компактного конструктора.* Компактный конструктор записи объявляется за счет указания имени записи, но без параметров. Компактный конструктор неявно имеет параметры, имена которых соответствуют компонентам записи, а значения аргументов, переданных конструктору, автоматически присваиваются данным компонентам. Тем не менее, внутри компактного конструктора можно изменить один или несколько аргументов до того, как их значения будут присвоены компонентам записи.

В следующем примере предыдущий канонический конструктор преобразуется в компактную форму:

```
// Использование компактной формы канонического конструктора для удаления
// любых ведущих и завершающих пробелов из строки имени.
public Employee {
  // Удалить любые ведущие и завершающие пробелы.
  name = name.trim();
}
```

Метод `trim()` вызывается для параметра `name` (который неявно объявлен компактным конструктором), а результат присваивается обратно параметру `name`. В конце компактного конструктора значение `name` автоматически присваивается соответствующему полю. Значение неявного параметра `idNum` в конце конструктора тоже присваивается соответствующему полю. Поскольку параметры неявно присваиваются соответствующим полям при завершении конструктора, нет необходимости явно инициализировать поля. Более того, это будет незаконно.

Ниже показана переделанная версия предыдущей программы, в которой демонстрируется компактный канонический конструктор:

```
// Использование компактного конструктора записи.

// Объявить запись для сотрудника.
record Employee(String name, int idNum) {

  // Использование компактного канонического конструктора для удаления
  // любых ведущих и завершающих пробелов из строки имени.
  public Employee {
    // Удалить любые ведущие и завершающие пробелы.
    name = name.trim();
  }
}

class RecordDemo2 {
  public static void main(String[] args) {
    Employee[] empList = new Employee[4];

    // Здесь имя не имеет ведущих или завершающих пробелов.
    empList[0] = new Employee("Doe, John", 1047);

    // Следующие три имени содержат ведущие и/или завершающие пробелы.
    empList[1] = new Employee("  Jones, Robert", 1048);
    empList[2] = new Employee("Smith, Rachel  ", 1049);
    empList[3] = new Employee("  Martin, Dave  ", 1050);
```

```
    // Использовать методы доступа к содержимому записи для отображения имен
    // и идентификационных номеров. Обратите внимание, что все ведущие и/или
    // завершающие пробелы с помощью конструктора были удалены
    // из компонента name.
    for(Employee e: empList)
      System.out.println("Идентификатором сотрудника " + e.name()
                         + " является " + e.idNum());
  }
}
```

Вот вывод:

```
Идентификатором сотрудника Doe, John является 1047
Идентификатором сотрудника Jones, Robert является 1048
Идентификатором сотрудника Smith, Rachel является 1049
Идентификатором сотрудника Martin, Dave является 1050
```

Как видите, имена были стандартизированы за счет удаления ведущих и завершающих пробелов. Чтобы удостовериться в необходимости вызова `trim()` для достижения такого результата, просто удалите компактный конструктор, перекомпилируйте и запустите программу. Ведущие и завершающие пробелы по-прежнему будут присутствовать в именах.

Объявление неканонического конструктора

Несмотря на то что канонического конструктора часто бывает достаточно, вы можете объявлять другие конструкторы. Основное требование заключается в том, что любой неканонический конструктор обязан сначала вызывать другой конструктор в записи через `this`. Часто будет вызываться канонический конструктор, что в конечном итоге гарантирует присваивание значений всем полям. Объявление неканонического конструктора позволяет создавать записи для особых случаев. Скажем, вы можете применять такой конструктор для создания записи, в которой одному или нескольким компонентам присваивается стандартное значение-заполнитель.

В следующей программе для `Employee` объявляется неканонический конструктор, который инициализирует имя известным значением, но присваивает полю `idNum` специальное значение `pendingID` (равное `-1` и обозначающее ожидающий идентификационный номер), указывая на то, что при создании записи значение идентификатора недоступно:

```
// Использование неканонического конструктора в записи.

// Объявить запись для сотрудника, в которой явно объявляются
// канонический и неканонический конструкторы.
record Employee(String name, int idNum) {
  // Использовать статическое поле в записи.
  static int pendingID = -1;
  // Использовать компактный канонический конструктор для удаления
  // любых ведущих и завершающих пробелов из строки имени.
  public Employee {
    // Удалить любые ведущие и завершающие пробелы.
    name = name.trim();
  }
```

```
  // Это неканонический конструктор. Обратите внимание,
  // что для создания записи он передает каноническому конструктору
  // не идентификационный номер, а pendingID.
  public Employee(String name) {
    this(name, pendingID);
  }
}

class RecordDemo3 {
  public static void main(String[] args) {
    Employee[] empList = new Employee[4];

    // Создать список сотрудников с использованием записи Employee.
    empList[0] = new Employee("Doe, John", 1047);
    empList[1] = new Employee("Jones, Robert", 1048);
    empList[2] = new Employee("Smith, Rachel", 1049);

    // Ожидающий идентификационный номер.
    empList[3] = new Employee("Martin, Dave");

    // Отобразить имена и идентификационные номера.
    for(Employee e: empList) {
      System.out.print("Идентификатором сотрудника " + e.name()
                       + " является ");
      if(e.idNum() == Employee.pendingID) System.out.println("ожидающий");
      else System.out.println(e.idNum());
    }
  }
}
```

Обратите особое внимание на создание записи для сотрудника "Martin, Dave" с использованием неканонического конструктора, который передает аргумент `name` каноническому конструктору, но в качестве значения `idNum` указывает `pendingID`. Такой прием позволяет создать запись-заполнитель, не указывая идентификационный номер. Еще один момент: значение `pendingID` объявлено в виде статического поля внутри `Employee`. Как объяснялось ранее, в объявлении записи поля экземпляра не разрешены, но применение статического поля допускается.

Как видите, в приведенной выше версии `Employee` объявляются как канонический, так и неканонический конструкторы, что полностью разрешено. В записи можно определять столько различных конструкторов, сколько нужно, при условии, что все они придерживаются правил, определенных для записи.

Важно подчеркнуть, что записи неизменяемы. В данном примере это означает, что при получении значения идентификатора для "Martin, Dave" старая запись должна быть заменена новой записью, которая содержит идентификационный номер. Изменить запись с целью обновления идентификатора невозможно. Неизменяемость записей является основной характеристикой.

Еще один пример конструктора записи

В завершение темы конструкторов записей рассмотрим еще один пример. Поскольку запись служит целям агрегирования данных, обычно конструктор

записи используется для организации проверки правильности или применимости аргумента. Скажем, перед созданием записи конструктору может потребоваться выяснить, не выходит ли значение за пределы допустимого диапазона, имеет ли оно неправильный формат или является ли непригодным для потребления по другим причинам. Если обнаружено недопустимое условие, тогда конструктор может создать стандартную запись или объект ошибки. Однако часто будет лучше, если конструктор сгенерирует исключение, что позволит пользователю записи сразу узнать об ошибке и принять меры по ее исправлению.

В предшествующих примерах записей `Employee` имена указывались с использованием общепринятого соглашения "фамилия, имя", например, "Doe, John", но не было никакого механизма для проверки или обеспечения того, чтобы имена действительно придерживались такого формата. В представленной далее версии компактного канонического конструктора организована ограниченная проверка того, что имя имеет формат "фамилия, имя". Цель достигается подтверждением того, что имя содержит одну и только одну запятую и минимум один (отличающийся от пробела) символ до и после запятой. Хотя в реальной программе потребовалась бы гораздо более полная и тщательная проверка, реализованной минимальной проверки вполне достаточно для иллюстрации той роли проверки, которую способен играть конструктор записи.

Итак, вот версия записи `Employee`, в которой компактный канонический конструктор генерирует исключение, если компонент `name` не соответствует минимальным критериям, требующимся для формата "фамилия, имя":

```
// Использование компактного канонического конструктора для удаления
// любых ведущих и завершающих пробелов в компоненте name. Кроме того,
// реализуется базовая проверка того, что строка, переданная в параметре
// name, представлена в требуемом формате "фамилия, имя".
public Employee {
  // Удалить любые ведущие и завершающие пробелы.
  name = name.trim();

  // Выполнить минимальную проверку того, что name
  // находится в формате "фамилия, имя".

  // Сначала удостовериться, что name содержит только одну запятую.
  int i = name.indexOf(',');  // искать разделяющую запятую
  int j = name.lastIndexOf(',');
  if(i != j) throw
    new IllegalArgumentException("Обнаружено несколько запятых.");

  // Затем удостовериться, что до и после запятой имеется
  // хотя бы по одному символу.
  if(i < 1 | name.length() == i+1) throw
    new IllegalArgumentException("Требуемый формат: фамилия, имя");
}
```

В случае применения такого конструктора следующий оператор по-прежнему корректен:

```
empList[0] = new Employee("Doe, John", 1047);
```

Тем не менее, три показанных ниже оператора недопустимы и приведут к генерации исключения:

```
// Отсутствует запятая между фамилией и именем.
empList[1] = new Employee("Jones Robert", 1048);

// Лишние запятые.
empList[1] = new Employee("Jones, ,Robert", 1048);

// Отсутствует фамилия.
empList[1] = new Employee(", Robert", 1048);
```

Кроме того, обдумайте способы улучшения способностей конструктора, касающихся проверки того, что имя представлено в надлежащем формате. Скажем, можете исследовать подход с использованием регулярного выражения (см. главу 31).

Создание методов получения для записи

Хотя и редко, но возникает необходимость в создании собственной реализации метода получения. В случае объявления метода получения его неявная версия больше не предоставляется. Одна из причин объявления собственного метода получения связана с возможностью генерации исключения, когда не удовлетворено какое-либо условие. Например, если запись содержит имя файла и URL-адрес, то метод получения для имени файла может генерировать исключение `FileNotFoundException` при отсутствии файла по указанному URL-адресу. Однако существует очень важное требование относительно создания методов получения: они должны придерживаться принципа неизменяемости записи. Таким образом, метод получения, возвращающий модифицированное значение, семантически сомнителен (и его следует избегать), даже если код будет синтаксически корректным.

К объявлению реализации метода получения применяется несколько правил. Метод получения обязан иметь такой же тип возвращаемого значения и имя, как и компонент, который он получает. Кроме того, он должен быть явно объявлен открытым. (Следовательно, для объявления метода получения в записи стандартной доступности недостаточно.) В объявлении метода получения не разрешена конструкция `throws`. Наконец, метод получения не должен быть ни обобщенным, ни статическим.

Лучшая альтернатива переопределения метода получения в случаях, когда нужно получить модифицированное значение компонента, предусматривает объявление отдельного метода со своим именем. Например, пусть для записи `Employee` необходимо извлечь из компонента `name` только фамилию. Использование для этой цели стандартного метода получения повлечет за собой изменение значения, полученного из компонента. Поступать так нежелательно, потому что тогда нарушится аспект неизменяемости записи. Тем не менее, можно объявить другой метод по имени `lastName()`, который будет возвращать только фамилию. Такой подход демонстрируется в следующей программе, где также применяется конструктор с проверкой формата из предыдущего раздела, чтобы гарантировать хранение имен в виде "фамилия, имя".

```
// Использование метода экземпляра в записи.

// В этой версии записи Employee предоставляется метод по имени lastName(),
// который возвращает из компонента name только фамилию.
// Она также содержит версию компактного конструктора, в которой
// производится проверка общепринятого формата "фамилия, имя".
record Employee(String name, int idNum) {

  // Использование компактного канонического конструктора для удаления
  // любых ведущих и завершающих пробелов в компоненте name. Кроме того,
  // реализуется базовая проверка того, что строка, переданная в параметре
  // name, представлена в требуемом формате "фамилия, имя".
  public Employee {
    // Удалить любые ведущие и завершающие пробелы.
    name = name.trim();

    // Выполнить минимальную проверку того, что name
    // находится в формате "фамилия, имя".

    // Сначала удостовериться, что name содержит только одну запятую.
    int i = name.indexOf(',');  // искать разделяющую запятую
    int j = name.lastIndexOf(',');
    if(i != j) throw
      new IllegalArgumentException("Обнаружено несколько запятых.");

    // Затем удостовериться, что до и после запятой имеется
    // хотя бы по одному символу.
    if(i < 1 | name.length() == i+1) throw
      new IllegalArgumentException("Требуемый формат: фамилия, имя");
  }

  // Метод экземпляра, который возвращает только фамилию, без имени.
  String lastName() {
    return name.substring(0, name.trim().indexOf(','));
  }
}

class RecordDemo4 {
  public static void main(String[] args) {
    Employee emp = new Employee("Jones, Robert", 1048);

    // Отобразить немодифицированный компонент name.
    System.out.println("Имя и фамилия сотрудника: " + emp.name());

    // Отобразить только фамилию.
    System.out.println("Фамилия сотрудника: " + emp.lastName());
  }
}
```

Вот вывод:

```
Имя и фамилия сотрудника: Jones, Robert
Фамилия сотрудника: Jones
```

В выводе видно, что неявный метод получения для компонента `name` возвращает имя без изменений. Метод экземпляра `lastName()` извлекает только фамилию. При таком подходе характеристика неизменяемости для записи `Employee` предохраняется, но одновременно предлагается удобный способ получения фамилии.

Сопоставление с образцом в операции instanceof

Традиционная форма операции instanceof обсуждалась в главе 13. Как там объяснялось, результатом instanceof будет true, если объект принадлежит к указанному типу или может быть к нему приведен. Начиная с версии JDK 16, в Java появилась вторая форма операции instanceof, которая поддерживает новую возможность *сопоставления с образцом*. В общих чертах сопоставление с образцом представляет собой механизм, который определяет, соответствует ли значение общей форме. Что касается instanceof, то сопоставление с образцом используется для проверки типа значения (которое должно быть ссылочным типом) на соответствие указанному типу. Такой шаблон называется *шаблоном типа*. Если шаблон совпадает, тогда *шаблонная переменная* получит ссылку на объект, соответствующий шаблону.

Ниже показана форма операции instanceof, поддерживающая сопоставление с образцом:

объектная-ссылка instanceof *тип шаблонная-переменная*

Если операция instanceof завершается успешно, тогда создается шаблонная переменная, содержащая ссылку на объект, который соответствует образцу. Если операция instanceof терпит неудачу, то шаблонная переменная создаваться не будет. Такая форма instanceof выполнится успешно, если объект, на который ссылается объектная ссылка, может быть приведен к указанному типу, когда статический тип объектной ссылки не является подтипом указанного типа.

Например, в показанном ниже фрагменте кода создается переменная типа Number по имени myOb, которая ссылается на объект Integer. (Вспомните, что Number — суперкласс для всех оболочек примитивных числовых типов.) Затем в коде применяется операция instanceof для подтверждения того, что объект, на который ссылается myOb, имеет тип Integer, каковым он и будет в данном примере. В результате создается объект iObj типа Integer, содержащий совпавшее значение.

```
Number myOb = Integer.valueOf(9);

// Использование версии instanceof, поддерживающей сопоставление с образцом
if(myOb instanceof Integer iObj) {
  // Здесь переменная iObj известна и находится в области видимости.
  System.out.println("iObj ссылается на целое число: " + iObj);
}
// Здесь переменная iObj не существует.
```

Как указано в комментариях, переменная iObj известна только внутри области видимости оператора if, но не за пределами if. Она также не была бы известной в операторе else при его наличии. Критически важно понимать, что шаблонная переменная iObj создается только в случае успешного сопоставления с образцом.

Основное преимущество формы instanceof, поддерживающей сопоставление с шаблоном, связано с тем, что она сокращает объем кода, который обычно требовался для традиционной формы instanceof. Скажем, рассмотрим следующую функционально эквивалентную версию предыдущего примера, где используется традиционная форма instanceof:

```
// Использование традиционной версии instanceof.
if(myOb instanceof Integer) {
  // Использовать для получения iObj явное приведение.
  Integer iObj = (Integer) myOb;
  System.out.println("iObj ссылается на целое число: " + iObj);
}
```

В случае применения традиционной формы instanceof для создания переменной iObj требуется отдельный оператор объявления и явное приведение. Форма instanceof, которая поддерживает сопоставление с шаблоном, упрощает процесс.

Шаблонные переменные в логических выражениях "И"

Операция instanceof может использоваться в логическом выражении "И". Однако необходимо помнить о том, что шаблонная переменная попадает в область видимости только после своего создания. Например, следующий оператор if выполняется успешно, только если myOb ссылается на объект Integer с неотрицательным значением. Обратите особое внимание на выражение в if:

```
if((myOb instanceof Integer iObj) && (iObj >= 0)) {  // нормально
  // myOb ссылается на объект Integer с неотрицательным значением.
  // ...
}
```

Шаблонная переменная iObj создается лишь в ситуации, когда левая часть операции && (содержащая instanceof) дает true. Обратите внимание, что iObj задействована также в правой части &&. Поступать подобным образом можно из-за применения короткозамкнутой формы логической операции "И", так что правая часть вычисляется, *только если* левая часть оказывается истинной. Таким образом, переменная iObj будет в области видимости в случае вычисления операнда в правой части. Тем не менее, если вы попытаетесь записать предыдущий оператор с использованием операции &, тогда на этапе компиляции возникнет ошибка, потому что переменная iObj не будет в области видимости, если левая часть даст false:

```
if((myOb instanceof Integer iObj) & (iObj >= 0)) {  // Ошибка!
  // myOb ссылается на объект Integer с неотрицательным значением.
  // ...
}
```

Вспомните, что операция & инициирует вычисление обеих частей выражения, но переменная iObj попадет в область видимости, только когда левая часть окажется true. Компилятор обнаруживает такую ошибку. Похожая ситуация возникает со следующим фрагментом:

```
int count = 10;
if((count < 100) && myOb instanceof Integer iObj) {  // нормально
  // myOb ссылается на объект Integer с неотрицательным значением,
  // а count меньше 100.
  iObj = count;
  // ...
}
```

Показанный фрагмент кода скомпилируется, т.к. блок `if` будет выполняться только тогда, когда обе части операции `&&` будут `true`. Таким образом, использовать `iObj` в блоке `if` разрешено. Однако если вы попытаетесь применить операцию `&`, а не `&&`, то на этапе компиляции возникнет ошибка:

```
if((count < 100) & myOb instanceof Integer iObj) {  // Ошибка!
```

В этом случае компилятор не в состоянии выяснить, появится ли переменная `iObj` в области видимости блока `if`, поскольку вовсе не факт, что правая часть операции `&` станет вычисляться. Еще один момент: логическое выражение не может вводить одну и ту же шаблонную переменную более одного раза. Например, создание той же самой шаблонной переменной в обоих операндах операции логического "И" будет ошибкой.

Сопоставление с образцом в других операторах

Несмотря на то что форма операции `instanceof`, которая поддерживает сопоставление с образцом, часто используется в операторе `if`, таким применением она отнюдь не ограничивается. Ее также можно задействовать в условной части операторов цикла. В качестве примера предположим, что вы обрабатываете коллекцию объектов, возможно содержащихся в массиве. Кроме того, в начале массива находится несколько строк, и вы хотите обработать именно их, но не остальные объекты в списке. В показанной далее кодовой последовательности задача решается с помощью цикла `for`, в условной части которого используется операция `instanceof` для подтверждения того, что объект в массиве является строкой, и для получения этой строки с целью обработки внутри тела цикла. Таким образом, сопоставление с образцом применяется для управления выполнением цикла `for` и получения следующего значения, подлежащего обработке.

```
Object[] someObjs = {
  new String("Альфа"),
  new String("Бета"),
  new String("Омега"),
  Integer.valueOf(10)
};
int i;
// Цикл повторяется до тех пор, пока элемент имеет тип String
// и не был достигнут конец массива.
for(i = 0; (someObjs[i] instanceof String str)  && (i < someObjs.length); i++)
{
  System.out.println("Обработка " + str);
  // ...
}
System.out.println("Количество начальных элементов в списке, являющихся
строками: " + i);
```

Ниже представлен вывод, получаемый в результате выполнения данного фрагмента:

```
Обработка Альфа
Обработка Бета
Обработка Омега
Количество начальных элементов в списке, являющихся строками: 3
```

Форма операции instanceof, которая поддерживает сопоставление с образцом, также может оказаться удобной в цикле while. Например, вот как превратить предыдущий цикл for в цикл while:

```
i = 0;
while((someObjs[i] instanceof String str) && (i < someObjs.length)) {
  System.out.println("Обработка " + str);
  i++;
}
```

Хотя задействовать операцию instanceof, поддерживающую сопоставление с образцом, в условной части цикла do формально допускается, такой сценарий строго ограничен из-за того, что шаблонную переменную нельзя использовать в теле цикла, поскольку она не будет находиться в области видимости, пока не выполнится операция instanceof.

Запечатанные классы и запечатанные интерфейсы

Начиная с версии JDK 17, разрешено объявлять класс, который может наследоваться только определенными подклассами. Такой класс называется *запечатанным*. До появления запечатанных классов наследование работало в стиле "все или ничего". Класс можно было либо расширять через любой подкласс, либо помечать как final, что полностью предотвращало наследование от него. Запечатанные классы находятся между этими двумя крайностями, потому что позволяют точно указывать, каким подклассам разрешено быть унаследованными от суперкласса. Аналогичным образом можно объявлять запечатанный интерфейс с указанием только реализующих его классов и/или расширяющих его интерфейсов. Вместе запечатанные классы и запечатанные интерфейсы обеспечивают значительно больший контроль над наследованием, что может быть особенно важно при проектировании библиотек классов.

Запечатанные классы

Чтобы объявить запечатанный класс, поместите sealed перед объявлением, а после имени класса укажите конструкцию permits с разрешенными подклассами. Как sealed, так и permits являются контекстно-чувствительными ключевыми словами, которые имеют особый смысл только в объявлении класса или интерфейса, но не за его пределами. Ниже показан простой пример запечатанного класса:

```
public sealed class MySealedClass permits Alpha, Beta {
  // ...
}
```

Запечатанный класс имеет имя `MySealedClass`. Он разрешает наследование от себя только двум подклассам: `Alpha` и `Beta`. При попытке наследования от `MySealedClass` любого другого класса возникнет ошибка на этапе компиляции.

Вот `Alpha` и `Beta` — два подкласса `MySealedClass`:

```
public final class Alpha extends MySealedClass {
  // ...
}
public final class Beta extends MySealedClass {
  // ...
}
```

Обратите внимание, что каждый из них помечен как `final`. Обычно подкласс запечатанного класса должен быть объявлен как `final`, `sealed` или `non-sealed`. Рассмотрим все варианты по очереди. Прежде всего, в этом примере каждый подкласс объявлен как `final`, т.е. подклассами `MySealedClass` будут только `Alpha` и `Beta`, а подклассы для них создавать нельзя. Следовательно, цепочка наследования заканчивается классами `Alpha` и `Beta`.

Чтобы отразить тот факт, что запечатанным является сам подкласс, его необходимо объявить как `sealed` и указать разрешенные подклассы. Например, вот версия `Alpha`, которая разрешает наследование подклассу `Gamma`:

```
public sealed class Alpha extends MySealedClass permits Gamma {
  // ...
}
```

Разумеется, класс `Gamma` должен быть объявлен как `sealed`, `final` или `non-sealed`.

Хотя выглядит это слегка неожиданным, но подкласс запечатанного класса можно распечатывать с помощью контекстно-чувствительного ключевого слова `non-sealed`, которое появилось в версии JDK 17. Оно разблокирует подкласс, позволяя любому другому классу наследоваться от него. Скажем, класс `Beta` можно было бы объявить так:

```
public non-sealed class Beta extends MySealedClass {
  // ...
}
```

Теперь любой класс может быть унаследован от `Beta`. Тем не менее, единственными прямыми подклассами `MySealedClass` остаются `Alpha` и `Beta`. Основная причина введения `non-sealed` связана с тем, чтобы позволить суперклассу указывать ограниченный набор прямых подклассов, которые обеспечивают базовый уровень четко определенной функциональности, но дать возможность свободно расширять эти подклассы.

Если имя класса присутствует в конструкции `permits` для запечатанного класса, тогда этот класс *обязан* напрямую расширять запечатанный класс, иначе возникнет ошибка на этапе компиляции. Таким образом, запечатанный

класс и его подклассы определяют взаимозависимую логическую единицу. Кроме того, объявлять как non-sealed класс, который не расширяет запечатанный класс, не допускается.

Ключевое требование запечатанного класса состоит в том, что каждый разрешенный для наследования от него подкласс должен быть доступным. Более того, если запечатанный класс содержится в именованном модуле, то каждый подкласс должен находиться в том же самом именованном модуле. В таком случае подкласс может располагаться не в том же пакете, что и запечатанный класс. Если запечатанный класс находится в неименованном модуле, тогда запечатанный класс и все разрешенные для наследования подклассы должны быть определены в одном и том же пакете.

В предыдущем обсуждении суперкласс MySealedClass и его подклассы Alpha и Beta были бы сохранены в отдельных файлах, поскольку все они — открытые классы. Однако запечатанный класс и его подклассы также допустимо хранить в одном файле (формально в единице компиляции) при условии, что подклассы имеют стандартный доступ к пакету. В ситуациях подобного рода для запечатанного класса конструкция permits не требуется. Например, в данном случае все три класса расположены в том же самом файле:

```
// Поскольку все классы находятся в одном файле,
// конструкция permits в MySealedClass не требуется.
public sealed class MySealedClass {
  // ...
}

final class Alpha extends MySealedClass {
  // ...
}

final class Beta extends MySealedClass {
  // ...
}
```

И последнее замечание: абстрактный класс может быть также и запечатанным. В этом отношении нет никаких ограничений.

Запечатанные интерфейсы

Запечатанный интерфейс объявляется аналогично запечатанному классу с применением sealed. Запечатанный интерфейс использует свою конструкцию permits для указания классов, которым позволяется его реализовывать, и/или интерфейсов, которым разрешено его расширять. Таким образом, отсутствующий в конструкции permits класс не может реализовывать запечатанный интерфейс, а интерфейс, не включенный в конструкцию permits, не может его расширять.

Вот простой пример запечатанного интерфейса, который разрешает его реализацию только классам Alpha и Beta:

```
public sealed interface MySealedIF permits Alpha, Beta {
  void myMeth();
}
```

Класс, реализующий запечатанный интерфейс, должен быть определен как final, sealed или non-sealed. Скажем, класс Alpha помечен как non-sealed, а Beta — как final:

```
public non-sealed class Alpha implements MySealedIF {
  public void myMeth() { System.out.println("Метод myMeth() в Alpha."); }
  // ...
}
public final class Beta implements MySealedIF {
  public void myMeth() { System.out.println("Метод myMeth() в Beta."); }
  // ...
}
```

Важный момент заключается в том, что любой класс, указанный в конструкции permits запечатанного интерфейса, *обязан* реализовывать интерфейс. Таким образом, запечатанный интерфейс и реализующие его классы формируют логическую единицу.

В запечатанном интерфейсе также допускается указывать, какие другие интерфейсы могут его расширять. Например, интерфейс MySealedIF разрешает расширять его интерфейсу MyIF:

```
// Обратите внимание на добавление MyIF в конструкцию permits.
public sealed interface MySealedIF permits Alpha, Beta, MyIF {
  void myMeth();
}
```

Поскольку MyIF присутствует в конструкции permits интерфейса MySealedIF, он должен быть помечен как non-sealed или sealed и обязан расширять MySealedIF:

```
public non-sealed interface MyIF extends MySealedIF {
  // ...
}
```

Как и следовало ожидать, класс может быть унаследован от запечатанного класса *и* реализовывать запечатанный интерфейс. Скажем, следующий класс Alpha унаследован от MySealedClass и реализует MySealedIF:

```
public non-sealed class Alpha extends MySealedClass implements MySealedIF {
  public void myMeth() { System.out.println("Метод myMeth() в Alpha."); }
  // ...
}
```

В предшествующих примерах классы и интерфейсы были объявлены открытыми и потому находились в собственных файлах. Тем не менее, как и в случае с запечатанными классами, запечатанный интерфейс и реализующие его классы (и расширяющие интерфейсы) также могут храниться в одном файле, если классы и интерфейсы имеют стандартный доступ к пакету. В случаях подобного рода для запечатанного интерфейса конструкция permits не требуется. Например, MySealedIF не содержит конструкцию permits, потому что классы Alpha и Beta объявлены в том же самом файле внутри неименованного модуля:

```
public sealed interface MySealedIF {
  void myMeth();
}

non-sealed class Alpha extends MySealedClass implements MySealedIF {
  public void myMeth() { System.out.println("Метод myMeth() в Alpha."); }
  // ...
}

final class Beta extends MySealedClass implements MySealedIF {
  public void myMeth() { System.out.println("Метод myMeth() в Beta."); }
  // ...
}
```

Последнее замечание: запечатанные классы и запечатанные интерфейсы больше всего пригодны разработчикам библиотек API, где производные классы и интерфейсы должны строго контролироваться.

Будущие направления развития

Начиная с версии JDK 12, выпуски Java нередко включали новое хорошо развитое расширение Java — *предварительные версии функциональных средств*, которые формально *пока* не входили в состав Java. Взамен программистам предоставлялось время поэкспериментировать с новым функциональным средством и при желании поделиться своими мыслями и мнениями до того, как оно станет постоянным. Такой процесс позволяет улучшить или оптимизировать новое функциональное средство, основываясь на фактическом использовании разработчиком. В результате функциональное средство, предлагаемое в виде предварительной версии, *подвержено* изменениям и даже может быть отозвано. Это означает, что предварительная версия функционального средства не должна применяться в коде, предназначенном для опубликования. В то же время ожидается, что большинство функциональных средств, предложенных для ознакомления в форме предварительных версий, в конечном итоге станут частью Java, возможно, после какого-то периода доработки. Предварительные версии демонстрируют направление будущего развития Java.

В состав JDK 17 входит предварительная версия одного функционального средства: сопоставление с образцом для `switch` (JEP 406), которое добавляет к оператору `switch` возможность сопоставления с образцом. Как было описано ранее в главе, сопоставление с образцом впервые появилось в расширении операции `instanceof`, введенном в версии JDK 16. Добавление сопоставления с образцом в `switch` продолжает данный процесс. Поскольку это предварительная версия функционального средства, которая может быть изменена, далее в книге она обсуждаться не будет.

Выпуски Java могут также содержать *инкубаторные модули*, которые предлагают предварительные версии нового API или инструмента, находящегося в стадии разработки. В точности как предварительные версии функциональных средств, средства из инкубаторных модулей могут изменяться, а то и

удаляться. Таким образом, нет никакой гарантии, что инкубаторный модуль в будущем официально станет частью Java. Функциональные средства из инкубаторных модулей дают разработчикам возможность поэкспериментировать с API или инструментом и оставить отзывы. В состав JDK 17 входят два инкубаторных модуля — Foreign Function and Memory API (JEP 412) и Vector API (JEP 414).

Важно подчеркнуть, что предварительные версии функциональных средств и инкубаторные модули могут присутствовать в любой версии Java, а потому нелишне будет следить за ними в каждой новой версии Java. Вы получаете возможность опробовать новое усовершенствование до того, как оно станет формальной частью Java. Однако важнее всего то, что предварительные версии функциональных средств и инкубаторные модули дают вам заблаговременную информацию относительно направления, в котором движется разработка Java.

ЧАСТЬ

II

Библиотека Java

ГЛАВА

18 Обработка строк

Краткий обзор обработки строк в Java был предложен в главе 7, а в текущей главе обработка строк рассматривается во всех подробностях. Как и в большинстве других языков программирования, строка в Java представляет собой последовательность символов. Но в отличие от ряда других языков, где строки реализованы в виде массивов символов, в Java строки представлены в форме объектов типа `String`.

Реализация строк как встроенных объектов позволяет Java предложить полный набор функций, делающих работу со строками удобной. Например, в Java есть методы для сравнения двух строк, поиска подстроки, объединения двух строк и изменения регистра букв в строке. Кроме того, объекты `String` могут быть сконструированы несколькими способами, что упрощает получение строки, когда это необходимо.

Несколько неожиданно, но в случае создания объекта `String` вы создаете строку, которую модифицировать нельзя, т.е. после создания объекта `String` изменять символы, составляющие строку, невозможно. Поначалу это может показаться серьезным ограничением, но ситуация вовсе не такая. Вы по-прежнему можете выполнять все типы строковых операций. Разница в том, что каждый раз, когда вам нужна измененная версия существующей строки, создается новый объект `String`, содержащий все модификации. Исходная строка остается неизмененной. Такой подход выбран из-за того, что фиксированные неизменяемые строки могут быть реализованы более эффективно, нежели изменяемые. В ситуациях, когда требуется модифицируемая строка, в Java предлагаются два варианта — `StringBuffer` и `StringBuilder`, которые хранят строки, допускающие изменение после их создания.

Классы `String`, `StringBuffer` и `StringBuilder` определены в пакете `java.lang` и потому автоматически доступны любым программам. Все они объявлены как `final`, т.е. ни для одного из них не может быть создан подкласс, что позволяет предпринимать определенные оптимизации, повышающие производительность обычных строковых операций. Все три реализуют интерфейс `CharSequence`.

И последнее замечание: неизменяемость строк в объектах типа `String` означает, что содержимое экземпляра `String` не может быть модифицировано после его создания. Но переменная, объявленная как ссылка на `String`, в любой момент может быть изменена с целью указания на другой объект `String`.

Конструкторы класса `String`

Класс `String` поддерживает несколько конструкторов. Для создания пустого экземпляра `String` вызывайте стандартный конструктор. Например, следующая строка кода создаст экземпляр `String`, не содержащий символов:

```
String s = new String();
```

Часто у вас будет возникать необходимость создавать строки с начальными значениями, для чего в классе `String` предусмотрено множество конструкторов. Чтобы создать строку, инициализированную массивом символов, используйте показанный ниже конструктор:

```
String(char[] chars)
```

Вот пример:

```
char[] chars = { 'a', 'b', 'c' };
String s = new String(chars);
```

В результате `s` инициализируется строкой `"abc"`.

С применением следующего конструктора в качестве инициализатора можно указывать поддиапазон массива символов:

```
String(char[] chars, int startIndex, int numChars)
```

В `startIndex` задается индекс, с которого начинается поддиапазон, а в `numChars` — количество используемых символов, например:

```
char[] chars = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

В итоге `s` инициализируется символами `cde`.

С помощью приведенного далее конструктора можно создать объект `String`, содержащий те же символы, что и другой объект `String`:

```
String(String strObj)
```

Здесь `strObj` представляет собой объект `String`. Рассмотрим пример:

```
// Создать объект String из другого объекта String.
class MakeString {
  public static void main(String[] args) {
    char[] c = {'J', 'a', 'v', 'a'};
    String s1 = new String(c);
    String s2 = new String(s1);

    System.out.println(s1);
    System.out.println(s2);
  }
}
```

Ниже показан вывод, генерируемый программой:

```
Java
Java
```

Как видите, `s1` и `s2` содержат ту же самую строку.

Хотя для типа char языка Java применяется 16 бит для представления основного набора символов Unicode, типичный формат строк в Интернете использует массивы 8-битных байтов, созданные из набора символов ASCII. С учетом такой распространенности 8-битных строк ASCII в классе String предлагаются конструкторы, которые инициализируют строку при наличии массива байтов:

```
String(byte[] chrs)
String(byte[] chrs, int startIndex, int numChars)
```

В chrs указывается массив байтов. Вторая форма позволяет задавать поддиапазон. В каждом конструкторе преобразование байтов в символы выполняется с применением стандартной кодировки символов, принятой для платформы. Использование этих конструкторов демонстрируется в следующей программе:

```
// Создание строки из поддиапазона массива байтов.
class SubStringCons {
  public static void main(String[] args) {
    byte[] ascii = {65, 66, 67, 68, 69, 70 };

    String s1 = new String(ascii);
    System.out.println(s1);

    String s2 = new String(ascii, 2, 3);
    System.out.println(s2);
  }
}
```

Программа генерирует такой вывод:

```
ABCDEF
CDE
```

Кроме того, определены расширенные версии конструкторов преобразования байтов в строки, которые позволяют указывать кодировку символов, определяющую способ преобразования байтов в символы. Тем не менее, чаще всего вы будете отдавать предпочтение стандартной кодировке, принятой для платформы.

На заметку! При каждом создании объекта String из массива содержимое последнего копируется. Если вы измените содержимое массива после создания строки, то объект String не изменится.

Создать экземпляр String из объекта StringBuffer можно с применением следующего конструктора:

```
String(StringBuffer strBufObj)
```

А создать экземпляр String из объекта StringBuilder можно с использованием такого конструктора:

```
String(StringBuilder strBuildObj)
```

Вот конструктор, который поддерживает расширенный набор символов Unicode:

```
String(int[] codePoints, int startIndex, int numChars)
```

В codePoints передается массив, содержащий кодовые точки Unicode. Результирующая строка создается из диапазона, который начинается с индекса startIndex и занимает numChars символов.

Вдобавок существуют конструкторы, позволяющие указывать набор символов.

На заметку! Обсуждение кодовых точек Unicode и способа их обработки в Java ищите в главе 19.

Длина строки

Длина строки — это количество содержащихся в ней символов. Чтобы получить значение длины строки, вызовите метод length():

```
int length()
```

Следующий кодовый фрагмент выводит 3, т.к. строка s содержит три символа:

```
char[] chars = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Специальные строковые операции

Поскольку строки являются распространенной и важной частью программирования, в синтаксис языка Java была добавлена специальная поддержка нескольких строковых операций. Такие операции включают автоматическое создание новых экземпляров String из строковых литералов, конкатенацию нескольких объектов String с помощью операции + и преобразование других типов данных в строковое представление. Для выполнения всех упомянутых функций доступны явные методы, но компилятор Java делает их автоматически для удобства программиста и большей ясности.

Строковые литералы

В предшествующих примерах было показано, каким образом явно создавать экземпляр String из массива символов с помощью операции new, но существует более простой способ, предусматривающий применение строкового литерала. Для каждого строкового литерала в программе автоматически создается объект String. Таким образом, строковый литерал можно использовать для инициализации объекта String. Скажем, следующий кодовый фрагмент создает две эквивалентные строки:

```
char[] chars = { 'a', 'b', 'c' };
String s1 = new String(chars);
String s2 = "abc";  // использовать строковый литерал
```

Поскольку для каждого строкового литерала создается объект String, строковый литерал разрешено применять в любом месте, где допускается

объект String. Например, методы можно вызывать прямо на строке в кавычках, как если бы она была ссылкой на объект. В операторе ниже для строки "abc" вызывается метод length(), что приводит к выводу значения 3:

```
System.out.println("abc".length());
```

Конкатенация строк

Обычно в Java не разрешено применять операции к объектам String. Единственным исключением из этого правила является операция +, которая выполняет конкатенацию двух строк, создавая в результате объект String. Кроме того, несколько операций + можно выстраивать в цепочку. Скажем, в следующем фрагменте реализована конкатенация трех строк, которая приводит к выводу строки "Ему 9 лет.":

```
String age = "9";
String s = "Ему " + age + " лет.";
System.out.println(s);
```

Один из случаев практического использования конкатенации строк касается создания очень длинных строк. Вместо распространения длинных строк на несколько строчек в исходном коде их можно разбить на более мелкие части и применять операцию + для конкатенации. Вот пример:

```
// Использование конкатенации для избавления от длинных строк.
class ConCat {
  public static void main(String[] args) {
    String longStr = "Такую очень длинную строку " +
      "пришлось бы разносить по нескольким " +
      "экранным строчкам. Но конкатенация " +
      "позволяет предотвратить это.";

    System.out.println(longStr);
  }
}
```

Конкатенация строк с другими типами данных

Конкатенацию строк можно выполнять с другими типами данных. Рассмотрим слегка отличающуюся версию ранее приведенного фрагмента кода:

```
int age = 9;
String s = "Ему " + age + " лет.";
System.out.println(s);
```

В новой версии переменная age имеет тип int, а не String, но результат остается прежним. Причина в том, что внутри объекта String значение int в age автоматически преобразуется в свое строковое представление, с которым выполняется конкатенация. Компилятор будет преобразовывать операнд в строковый эквивалент всякий раз, когда другой операнд операции + является экземпляром String.

Однако будьте осторожны при смешивании других видов операций с выражениями конкатенации строк, иначе полученные результаты могут удивить. Взгляните на следующий фрагмент кода:

```
String s = "Четыре: " + 2 + 2;
System.out.println(s);
```

Вывод будет таким:

```
Четыре: 22
```

а не тем, что можно было ожидать:

```
Четыре: 4
```

И вот почему. Приоритеты операций приводят к тому, что конкатенация строки "Четыре" со строковым эквивалентом значения 2 выполняется первой. Затем результат объединяется с еще одним строковым эквивалентом 2. Чтобы сначала выполнилось сложение целых чисел, необходимо использовать круглые скобки:

```
String s = "Четыре: " + (2 + 2);
```

Теперь s будет содержать строку "Четыре: 4".

Преобразование в строку и toString()

Один из способов преобразования данных в строковое представление предусматривает вызов одной из перегруженных версий метода преобразования строк valueOf(), определенного в классе String. Метод valueOf() перегружен для всех примитивных типов и для типа Object. Для примитивных типов valueOf() возвращает строку, содержащую удобочитаемый эквивалент значения, с которым оно вызывается. Для объектов valueOf() вызывает метод toString() объекта. Мы более подробно рассмотрим valueOf() позже в главе, а пока обсудим метод toString(), потому что он является средством, с помощью которого можно определить строковое представление для объектов создаваемых классов.

Каждый класс реализует метод toString(), т.к. он определен в Object. Тем не менее, стандартной реализации toString() редко бывает достаточно. Для наиболее важных создаваемых классов вы наверняка решите переопределить метод toString() и предоставить собственные строковые представления. К счастью, делается это легко. Метод toString() имеет следующую общую форму:

```
String toString()
```

Чтобы реализовать toString(), просто возвратите объект String, содержащий удобочитаемую строку, которая надлежащим образом описывает объект вашего класса.

Переопределяя метод toString() для создаваемых вами классов, вы позволяете им полностью интегрироваться в среду программирования Java. Например, их можно применять в операторах print() и println(), а также в выражениях конкатенации. В следующей программе это демонстрируется на примере переопределения метода toString() для класса Box:

```
// Переопределение метода toString() для класса Box.
class Box {
  double width;
  double height;
  double depth;

  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
  public String toString() {
    return "Размеры коробки: " + width + " на " +
      depth + " на " + height + ".";
  }
}
class toStringDemo {
  public static void main(String[] args) {
    Box b = new Box(10, 12, 14);
    String s = "Коробка b: " + b;  // конкатенация с объектом Box

    System.out.println(b);         // преобразование Box в строку
    System.out.println(s);
  }
}
```

Вот вывод, сгенерированный программой:

```
Размеры коробки: 10.0 на 14.0 на 12.0
Коробка b: Размеры коробки 10.0 на 14.0 на 12.0
```

Как видите, метод `toString()` класса `Box` вызывается автоматически, когда объект `Box` используется в выражении конкатенации или при вызове `println()`.

Извлечение символов

Класс `String` поддерживает несколько способов извлечения символов из объекта `String`, часть которых исследуются ниже. Хотя символы, составляющие строку в объекте `String`, не могут быть проиндексированы, как если бы они были массивом символов, многие методы класса `String` при своей работе задействуют индекс (или смещение) в строке. Подобно массивам индексы строк начинаются с нуля.

charAt()

Для извлечения из строки одного символа на него можно сослаться напрямую с помощью метода `charAt()`, общая форма которого показана ниже:

```
char charAt(int where)
```

Здесь `where` — индекс символа, который необходимо извлечь. Значение `where` должно быть неотрицательным и указывать местоположение в строке. Метод `charAt()` возвращает символ в указанной позиции.

Например, следующий код присваивает ch значение b:

```
char ch;
ch = "abc".charAt(1);
```

getChars()

Когда нужно извлечь более одного символа за раз, можно применить метод getChars(). Вот его общая форма:

```
void getChars(int sourceStart, int sourceEnd, char[] target, int targetStart)
```

В sourceStart указывается индекс начала подстроки, а в sourceEnd — индекс, следующий после конца желаемой подстроки. Таким образом, подстрока содержит символы от sourceStart до sourceEnd-1. В target передается массив, который получит символы. В targetStart задается индекс внутри target, по которому будет скопирована подстрока. Необходимо позаботиться о том, чтобы массив target имел достаточный размер, позволяющий вместить количество символов в указанной подстроке. Работа метода getChars() демонстрируется в приведенной ниже программе:

```
class getCharsDemo {
  public static void main(String[] args) {
    String s = "Строка для демонстрации работы метода getChars().";
    int start = 11;
    int end = 15;
    char[] buf = new char[end - start];
    s.getChars(start, end, buf, 0);
    System.out.println(buf);
  }
}
```

Вот вывод, генерируемый программой:

```
демо
```

getBytes()

Существует альтернативная версия метода getChars(), которая сохраняет символы в массиве байтов. Метод называется getBytes() и использует стандартное преобразование символов в байты, обеспечиваемое платформой. Вот его простейшая форма:

```
byte[] getBytes()
```

Доступны и другие формы getBytes(). Метод getBytes() наиболее полезен при экспортировании значения String в среду, где не поддерживаются 16-битные символы Unicode.

toCharArray()

Преобразовать все символы объекта String в массив символов проще всего путем вызова метода toCharArray(), который возвращает массив символов для целой строки и имеет следующую общую форму:

```
char[] toCharArray()
```

Метод `toCharArray()` предоставлен ради удобства, поскольку для достижения того же результата можно применять `getChars()`.

Сравнение строк

В состав класса `String` входит несколько методов, предназначенных для сравнения строк или подстрок внутри строк. Далее рассматриваются некоторые из них.

`equals()` и `equalsIgnoreCase()`

Для сравнения двух строк на предмет равенства используйте метод `equals()` с такой общей формой:

```
boolean equals(Object str)
```

В `str` указывается объект `String`, который сравнивается с вызывающим объектом `String`. Метод `equals()` возвращает `true`, если строки содержат одни и те же символы в одном и том же порядке, и `false` в противном случае. Сравнение чувствительно к регистру.

Чтобы выполнить сравнение с игнорированием различий в регистре, применяйте метод `equalsIgnoreCase()`. При сравнении двух строк он считает, что A–Z совпадают с a–z. Вот его общая форма:

```
boolean equalsIgnoreCase(String str)
```

Здесь `str` — объект `String`, который сравнивается с вызывающим объектом `String`. Метод `equalsIgnoreCase()` возвращает `true`, если строки содержат одни и те же символы, следующие в одинаковом порядке, и `false` в противном случае.

Ниже приведен пример, демонстрирующий работу `equals()` и `equalsIgnoreCase()`:

```
// Демонстрация работы equals() и equalsIgnoreCase().
class equalsDemo {
  public static void main(String[] args) {
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = "Good-bye";
    String s4 = "HELLO";
    System.out.println("Сравнение строк " + s1 + " и " + s2
                       + " с помощью equals() -> " + s1.equals(s2));
    System.out.println("Сравнение строк " + s1 + " и " + s3
                       + " с помощью equals() -> " + s1.equals(s3));
    System.out.println("Сравнение строк " + s1 + " и " + s4
                       + " с помощью equals() -> " + s1.equals(s4));
    System.out.println("Сравнение строк " + s1 + " и " + s4
                       + " с помощью equalsIgnoreCase() -> "
                       + s1.equalsIgnoreCase(s4));
  }
}
```

Вот вывод, генерируемый программой:

```
Сравнение строк Hello и Hello с помощью equals() -> true
Сравнение строк Hello и Good-bye с помощью equals() -> false
Сравнение строк Hello и HELLO с помощью equals() -> false
Сравнение строк Hello и HELLO с помощью equalsIgnoreCase() -> true
```

regionMatches()

Метод `regionMatches()` сравнивает определенную область внутри одной строки с определенной областью в другой строке. Существует перегруженная форма, позволяющая игнорировать регистр при таком сравнении. Далее показаны общие формы двух методов:

```
boolean regionMatches(int startIndex, String str2,
                      int str2StartIndex, int numChars)
boolean regionMatches(boolean ignoreCase,
                      int startIndex, String str2,
                      int str2StartIndex, int numChars)
```

В обеих версиях в `startIndex` указывается индекс, с которого начинается область внутри вызывающего объекта `String`. Сравниваемый объект `String` передается в `str2`. Индекс, с которого начнется сравнение в `str2`, задается в `str2StartIndex`. Длина сравниваемой подстроки указывается в `numChars`. Если `ignoreCase` во второй версии имеет значение `true`, то регистр символов игнорируется, и учитывается в противном случае.

startsWith() и endsWith()

В классе `String` определены два метода, которые являются в той или иной степени специализированными формами `regionMatches()`. Метод `startWith()` определяет, начинается ли объект `String` с указанной строки, а метод `endWith()` выясняет, заканчивается ли объект `String` заданной строкой. Вот их общие формы:

```
boolean startsWith(String str)
boolean endsWith(String str)
```

В `str` передается проверяемый объект `String`. В случае совпадения возвращается `true`, а иначе `false`. Например, `"Foobar".endsWith("bar")` и `"Foobar".startsWith("Foo")` дают `true`.

Представленная далее вторая форма `startsWith()` позволяет указать начальную позицию:

```
boolean startsWith(String str, int startIndex)
```

В `startIndex` задается индекс в вызывающей строке, с которого начнется поиск. Например, `"Foobar".startsWith("bar", 3)` возвращает `true`.

equals() или ==

Важно понимать, что метод `equals()` и операция == выполняют два разных действия. Как объяснялось выше, метод `equals()` сравнивает символы

внутри объекта String. Операция == сравнивает две объектные ссылки, чтобы определить, ссылаются ли они на один и тот же экземпляр. В следующей программе показано, что два разных объекта String могут содержать одни и те же символы, но ссылки на эти объекты не будут считаться равными:

```
// equals() или ==.
class EqualsNotEqualTo {
  public static void main(String[] args) {
    String s1 = "Hello";
    String s2 = new String(s1);
    System.out.println("Сравнение строк " + s1 + " и " + s2
                       + " с помощью equals() -> " + s1.equals(s2));
    System.out.println("Сравнение строк " + s1 + " и " + s4
                       + " с помощью == -> " + (s1 == s2));
  }
}
```

Переменная s1 ссылается на экземпляр String, созданный с помощью строки "Hello". Объект, на который ссылается переменная s2, создается с применением s1 в качестве инициализатора. Таким образом, содержимое двух объектов String идентично, но это разные объекты. Другими словами, s1 и s2 не ссылаются на тот же самый объект, так что результатом операции == будет false:

```
Сравнение строк Hello и Hello с помощью equals() -> true
Сравнение строк Hello и Hello с помощью == -> false
```

compareTo()

Часто недостаточно просто знать, идентичны ли две строки. В приложениях сортировки должно быть известно, какая строка *меньше*, *равна* или *больше* следующей. Одна строка меньше другой, если она расположена раньше в словарном порядке. Одна строка больше другой, если она расположена позже в словарном порядке. Для такой цели предназначен метод compareTo(), который определен в интерфейсе Comparable<T>, реализуемом String. Вот его общая форма:

```
int compareTo(String str)
```

В str указывается объект String, сравниваемый с вызывающим объектом String. Результат сравнения возвращается и интерпретируется, как показано в табл. 18.1.

Таблица 18.1. Результат сравнения, возвращаемый методом compareTo()

Значение	Описание
Меньше нуля	Вызывающая строка меньше str
Больше нуля	Вызывающая строка больше str
Ноль	Две строки равны

Далее приведен пример программы, которая сортирует массив строк. Для определения порядка при пузырьковой сортировке в ней используется метод compareTo():

```
// Пузырьковая сортировка строк.
class SortString {
  static String[] arr = {
    "Now", "is", "the", "time", "for", "all", "good", "men",
    "to", "come", "to", "the", "aid", "of", "their", "country"
  };
  public static void main(String[] args) {
    for(int j = 0; j < arr.length; j++) {
      for(int i = j + 1; i < arr.length; i++) {
        if(arr[i].compareTo(arr[j]) < 0) {
          String t = arr[j];
          arr[j] = arr[i];
          arr[i] = t;
        }
      }
      System.out.println(arr[j]);
    }
  }
}
```

Программа выводит список слов:

```
Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to
```

В выводе видно, что метод compareTo() принимает во внимание прописные и строчные буквы. Слово "Now" находится раньше всех остальных, т.к. оно начинается с прописной буквы, а потому имеет меньший код в наборе символов ASCII.

Чтобы игнорировать различия в регистре при сравнении двух строк, применяйте метод compareToIgnoreCase():

```
int compareToIgnoreCase(String str)
```

Метод compareToIgnoreCase() возвращает те же результаты, что и compareTo(), но игнорирует различия в регистре. После замены им вызова compareTo() в предыдущей программе слово "Now" больше не будет первым.

Поиск в строках

Класс `String` предлагает два метода, которые позволяют искать в строке указанный символ или подстроку:

- `indexOf()` — ищет первое вхождение символа или подстроки;
- `lastIndexOf()` — ищет последнее вхождение символа или подстроки.

Указанные два метода имеют несколько перегруженных версий. Во всех случаях они возвращают индекс, по которому был найден символ или подстрока, либо `-1`, если символ или подстрока не обнаружены.

Для поиска первого вхождения символа используйте

```
int indexOf(int ch)
```

Для поиска последнего вхождения символа применяйте

```
int lastIndexOf(int ch)
```

Искомый символ указывается в `ch`.

Для поиска первого или последнего вхождения подстроки используйте

```
int indexOf(String str)
int lastIndexOf(String str)
```

Искомая подстрока указывается в `str`.

Начальную позицию для поиска можно задавать в случае применения следующих форм методов:

```
int indexOf(int ch, int startIndex)
int lastIndexOf(int ch, int startIndex)
int indexOf(String str, int startIndex)
int lastIndexOf(String str, int startIndex)
```

В `startIndex` указывается индекс, с которого начинается поиск. Метод `indexOf()` выполняет поиск от `startIndex` до конца строки. Метод `lastIndexOf()` выполняет поиск от `startIndex` до нулевой позиции.

Ниже демонстрируется использование различных методов, работающих с индексами, для поиска внутри строки:

```
// Демонстрация работы indexOf() и lastIndexOf().
class indexOfDemo {
  public static void main(String[] args) {
    String s = "Now is the time for all good men " +
               "to come to the aid of their country.";
    System.out.println(s);
    System.out.println("indexOf(t) = " + s.indexOf('t'));
    System.out.println("lastIndexOf(t) = " + s.lastIndexOf('t'));
    System.out.println("indexOf(the) = " + s.indexOf("the"));
    System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
    System.out.println("indexOf(t, 10) = " + s.indexOf('t', 10));
    System.out.println("lastIndexOf(t, 60) = " + s.lastIndexOf('t', 60));
    System.out.println("indexOf(the, 10) = " + s.indexOf("the", 10));
    System.out.println("lastIndexOf(the, 60) = " + s.lastIndexOf("the", 60));
  }
}
```

Вот вывод, генерируемый программой:

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

Модификация строк

Поскольку объекты `String` являются неизменяемыми, всякий раз, когда нужно модифицировать строку, ее понадобится либо скопировать в `StringBuffer` или `StringBuilder`, либо использовать метод `String`, который создает новую копию строки с готовыми изменениями. Примеры таких методов описаны ниже.

substring()

Извлечь подстроку можно с применением метода `substring()`, который имеет две формы. Первая форма выглядит так:

```
String substring(int startIndex)
```

В `startIndex` указывается индекс, с которого будет начинаться подстрока. Первая форма `substring()` возвращает копию подстроки, которая начинается с позиции `startIndex` и продолжается до конца вызывающей строки.

Вторая форма `substring()` позволяет задавать индексы начала и конца подстроки:

```
String substring(int startIndex, int endIndex)
```

Здесь в `startIndex` указывается начальный индекс, а в `endIndex` — точка останова. Возвращаемая строка содержит все символы от начального индекса до конечного индекса, не включая его.

В следующей программе метод `substring()` используется для замены всех вхождений подстроки в строке другой строкой:

```
// Демонстрация замены подстроки.
class StringReplace {
  public static void main(String[] args) {
    String org = "This is a test. This is, too.";
    String search = "is";
    String sub = "was";
    String result = "";
    int i;

    do{  // заменить все соответствующие подстроки
      System.out.println(org);
      i = org.indexOf(search);
      if(i != -1) {
```

```
        result = org.substring(0, i);
        result = result + sub;
        result = result + org.substring(i + search.length());
        org = result;
      }
    } while(i != -1);
  }
}
```

Вот вывод, который генерирует программа:

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```

concat()

С применением метода `concat()` можно выполнять конкатенацию двух строк:

```
String concat(String str)
```

Метод `concat()` создает новый объект, содержащий вызывающую строку с добавленным в конец содержимым `str`. Его действие аналогично операции +. Например, следующий код помещает в `s2` строку `"onetwo"`:

```
String s1 = "one";
String s2 = s1.concat("two");
```

Он генерирует такой же результат, как и показанный далее код:

```
String s1 = "one";
String s2 = s1 + "two";
```

replace()

Метод `replace()` имеет две формы. Первая форма заменяет все вхождения одного символа в вызывающей строке другим символом:

```
String replace(char original, char replacement)
```

В `original` указывается символ, который должен быть заменен символом, заданным в `replacement`. Метод возвращает результирующую строку. Скажем, следующий код помещает в `s` строку `"Hewwo"`:

```
String s = "Hello".replace('l', 'w');
```

Вторая форма `replace()` заменяет одну последовательность символов другой:

```
String replace(CharSequence original, CharSequence replacement)
```

trim() и strip()

Метод trim() возвращает копию вызывающей строки, из которой удалены все ведущие и конечные пробелы. Применительно к этому методу пробелами считаются символы с кодами 32 и меньше. Он имеет следующую общую форму:

```
String trim()
```

Вот пример:

```
String s = "   Hello World   ".trim();
```

Значением переменной s становится строка "Hello World".

Метод trim() весьма полезен при обработке пользовательских команд. Например, показанная ниже программа запрашивает у пользователя название штата и затем отображает столицу этого штата. Метод trim() используется в ней для удаления любых начальных или конечных пробелов, которые могли быть случайно введены пользователем.

```
// Использование trim() для обработки команд.
import java.io.*;

class UseTrim {
  public static void main(String[] args)
    throws IOException
  {
    // Создать экземпляр BufferedReader, используя System.in.
    BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in, System.console().charset()));
    String str;

    System.out.println("Для завершения введите 'стоп'.");
    System.out.println("Введите название штата: ");
    do {
      str = br.readLine();
      str = str.trim();  // удалить пробельные символы

      if(str.equals("Иллинойс"))
        System.out.println("Столица - Спрингфилд.");
      else if(str.equals("Миссури"))
        System.out.println("Столица - Джефферсон-Сити.");
      else if(str.equals("Калифорния"))
        System.out.println("Столица - Сакраменто.");
      else if(str.equals("Вашингтон"))
        System.out.println("Столица - Олимпия.");
      // ...
    } while(!str.equals("стоп"));
  }
}
```

Начиная с версии JDK 11, в Java также предлагаются методы strip(), stripLeading() и stripTrailing(). Метод strip() удаляет все пробельные символы (как определено в Java) в начале и в конце вызывающей строки и возвращает результат. К таким пробельным символам относятся помимо прочих пробелы, символы табуляции, возврат каретки и перевод строки.

Методы stripLeading() и stripTrailing() удаляют пробельные символы соответственно в начале или в конце вызывающей строки и возвращают результат. В JDK 15 добавлены методы stripIndent(), которые удаляют лишние пробельные символы, сохраняя осмысленные отступы, и translateEscapes(), которые заменяют управляющие последовательности их символьными эквивалентами.

Преобразование данных с использованием valueOf()

Метод valueOf() преобразует данные из своего внутреннего формата в удобочитаемую форму. Это статический метод, перегруженный в String для всех встроенных типов Java, чтобы каждый тип можно было правильно преобразовывать в строку. Метод valueOf() также перегружен для типа Object, так что в качестве аргумента может передаваться и объект любого создаваемого вами класса. (Вспомните, что Object является суперклассом для всех классов.) Вот несколько форм valueOf():

```
static String valueOf(double num)
static String valueOf(long num)
static String valueOf(Object ob)
static String valueOf(char[] chars)
```

Как обсуждалось ранее, метод valueOf() можно вызывать, когда требуется строковое представление какого-то другого типа данных. Метод valueOf() можно вызывать напрямую с любым типом данных и получить приемлемое строковое представление. Все простые типы преобразуются в их обычные строковые представления. Любой объект, передаваемый методу valueOf(), возвратит результат вызова метода toString() объекта. На самом деле можно просто вызвать toString() напрямую и получить тот же результат.

Для большинства массивов valueOf() возвращает довольно загадочную строку, указывающую на то, что это массив определенного типа. Однако для символьных массивов создается объект String, содержащий символы из массива. Существует специальная версия valueOf(), которая позволяет указать подмножество символьного массива и имеет такую общую форму:

```
static String valueOf(char[] chars, int startIndex, int numChars)
```

Здесь chars — массив, содержащий символы, startIndex — индекс, с которого начинается нужная подстрока, и numChars — длина подстроки.

Изменение регистра символов внутри строк

Метод toLowerCase() преобразует все символы строки из верхнего регистра в нижний. Метод toUpperCase() преобразует все символы строки из нижнего регистра в верхний. Неалфавитные символы, такие как цифры, остаются незатронутыми. Ниже приведены их самые простые формы:

```
String toLowerCase()
String toUpperCase()
```

Оба метода возвращают объект `String`, который содержит эквивалент вызывающей строки в верхнем или нижнем регистре. В обоих случаях преобразование управляется стандартной локалью.

Вот пример, в котором применяются методы `toLowerCase()` и `toUpperCase()`:

```
// Демонстрация использования toUpperCase() и toLowerCase().
class ChangeCase {
  public static void main(String[] args)
  {
    String s = "This is a test.";

    System.out.println("Первоначальная строка: " + s);

    String upper = s.toUpperCase();
    String lower = s.toLowerCase();

    System.out.println("Строка в верхнем регистре: " + upper);
    System.out.println("Строка в нижнем регистре: " + lower);
  }
}
```

Программа генерирует следующий вывод:

```
Первоначальная строка: This is a test.
Строка в верхнем регистре: THIS IS A TEST.
Строка в нижнем регистре: this is a test.
```

Еще один момент: предоставляются также перегруженные версии методов `toLowerCase()` и `toUpperCase()`, которые позволяют указывать объект `Locale` для управления преобразованием. Указание локали в некоторых случаях может оказаться очень важным, помогая интернационализировать приложение.

Соединение строк

Метод `join()` используется для соединения двух и более строк с отделением строк друг от друга с помощью разделителя, такого как пробел или запятая. Он имеет две формы, первая из которых показана ниже:

```
static String join(CharSequence delim, CharSequence . . . strs)
```

В `delim` указывается разделитель, который применяется для отделения символьных последовательностей, заданных в `strs`. Поскольку класс `String` реализует интерфейс `CharSequence`, то в `strs` можно передавать список строк. (Интерфейс `CharSequence` описан в главе 19.) Первая форма `join()` демонстрируется в следующей программе:

```
// Демонстрация использования метода join(), определенного в String.
class StringJoinDemo {
  public static void main(String[] args) {

    String result = String.join(" ", "Alpha", "Beta", "Gamma");
```

```
    System.out.println(result);
    result = String.join(", ", "John", "ID#: 569",
                         "E-mail: John@HerbSchildt.com");
    System.out.println(result);
  }
}
```

Вот вывод:

```
Alpha Beta Gamma
John, ID#: 569, E-mail: John@HerbSchildt.com
```

При первом вызове `join()` между строками вставляется пробел. Во втором вызове разделителем является запятая, за которой следует пробел, т.е. разделитель не обязательно должен быть одним символом.

Вторая форма `join()` позволяет соединить список строк, полученных из объекта, который реализует интерфейс `Iterable`. Среди прочего интерфейс `Iterable` реализуется классами в инфраструктуре Collections Framework, рассматриваемой в главе 20, а сам интерфейс `Iterable` обсуждается в главе 19.

Дополнительные методы класса `String`

Помимо упомянутых ранее методов в классе `String` есть и много других, часть которых кратко описана в табл. 18.2.

Таблица 18.2. Некоторые дополнительные методы класса `String`

Метод	Описание
`int codePointAt(int i)`	Возвращает кодовую точку Unicode в позиции, указанной с помощью `i`
`int codePointBefore(int i)`	Возвращает кодовую точку Unicode в позиции, которая предшествует указанной с помощью `i`
`int codePointCount( int start, int end)`	Возвращает количество кодовых точек в части вызывающего объекта `String`, расположенной между `start` и `end-1`
`boolean contains( CharSequence str)`	Возвращает `true`, если вызывающий объект содержит строку, указанную в `str`, или `false` в противном случае
`boolean contentEquals( CharSequence str)`	Возвращает `true`, если вызывающий объект содержит ту же строку, которая указана в `str`, или `false` в противном случае
`boolean contentEquals( StringBuffer str)`	Возвращает `true`, если вызывающий объект содержит ту же строку, которая указана в `str`, или `false` в противном случае

Продолжение табл. 18.2

Метод	Описание
`static String format( String fmtstr, Object ... args)`	Возвращает строку, сформатированную, как указано в `fmtstr`. (За дополнительными деталями обращайтесь в главу 20.)
`static String format( Locale loc, String fmtstr, Object ... args)`	Возвращает строку, сформатированную, как указано в `fmtstr`. Форматирование управляется локалью, заданной в `loc`. (За дополнительными деталями обращайтесь в главу 20.)
`String formatted( Object ... args)`	Возвращает строку, сформатированную, как указано вызывающей строкой, применительно к `args`. (За дополнитель-ными деталями обращайтесь в главу 21.)
`String indent(int num)`	Когда `num` имеет положительное значение, делает отступ каждой строчки в вызываемой строке на `num` пробелов. Когда `num` имеет отрицательное значение, в каждой строчке удаляется `num` начальных пробельных символов, если это возможно. Иначе при отрицательном `num` ведущие пробелы удаляются до тех пор, пока не встретится первый непробельный символ. Во всех случаях, включая нулевое значение `num`, каждая строчка будет заканчиваться символом новой строки. Возвращает результирующую строку
`boolean isEmpty()`	Возвращает `true`, если вызывающая строка не содержит символов и имеет нулевую длину, или `false` в противном случае
`Stream<String> lines()`	Разбивает строку на отдельные строчки на основе символов возврата каретки и перевода строки и возвращает объект `Stream`, содержащий строчки
`boolean matches( string regExp)`	Возвращает `true`, если вызывающая строка соответствует регулярному выражению, переданному в `regExp`, или `false` в противном случае
`int offsetByCodePoints( int start, int num)`	Возвращает индекс внутри вызывающей строки, который на `num` кодовых точек превышает начальный индекс, указанный в `start`

Окончание табл. 18.2

Метод	Описание
`String replaceFirst( String regExp, String newStr)`	Возвращает строку, в которой первая подстрока, соответствующая регулярному выражению `regExp`, заменяется той, что указана в `newStr`
`String replaceAll( String regExp, String newStr)`	Возвращает строку, в которой все подстроки, соответствующие регулярному выражению `regExp`, заменяются строкой, указанной в `newStr`
`String[] split( String regExp)`	Разбивает вызывающую строку на части и возвращает массив, который содержит результат. Каждая часть разграничивается с помощью регулярного выражения, переданного в `regExp`
`String[] split( String regExp, int max)`	Разбивает вызывающую строку на части и возвращает массив, который содержит результат. Каждая часть разграничивается с помощью регулярного выражения, переданного в `regExp`. Количество частей указывается в `max`. Если `max` имеет отрицательное значение, тогда вызывающая строка разбивается полностью. Если же `max` является ненулевым, то последний элемент в возвращаемом массиве содержит остаток вызывающей строки. Если значение `max` равно нулю, тогда вызывающая строка разбивается полностью, но завершающие пустые строки в результат не включаются
`CharSequence subSequence( int startIndex, int stopIndex)`	Возвращает подстроку вызывающей строки, начинающуюся с индекса `startIndex` и заканчивающуюся индексом `stopIndex`. Этот метод требуется для интерфейса `CharSequence`, который реализуется классом `String`
`<R> R transform( Function<? super String, ? extends R> func)`	Выполняет функцию, указанную в `func`, в отношении вызывающей строки и возвращает результат

Обратите внимание, что некоторые методы работают с регулярными выражениями. Регулярные выражения описаны в главе 31. Еще один момент: начиная с версии JDK 12, класс `String` реализует интерфейсы `Constable` и `ConstantDesc`.

Класс StringBuffer

Класс StringBuffer поддерживает модифицируемую строку. Как вам уже известно, String представляет неизменяемую последовательность символов фиксированной длины. Напротив, StringBuffer представляет расширяемые и записываемые последовательности символов. Объект StringBuffer допускает вставку символов и подстрок в середину или добавление их в конец. Объект StringBuffer будет автоматически увеличиваться, освобождая место для таких дополнений, и часто имеет больше предварительно выделенных символов, чем на самом деле необходимо, чтобы оставить место для роста.

Конструкторы класса StringBuffer

В классе StringBuffer определены четыре конструктора:

```
StringBuffer()
StringBuffer(int size)
StringBuffer(String str)
StringBuffer(CharSequence chars)
```

Стандартный конструктор (не имеющий параметров) резервирует место для 16 символов без перераспределения. Вторая версия конструктора принимает целочисленный аргумент, который явно устанавливает размер буфера. Третья версия принимает аргумент String, который устанавливает начальное содержимое объекта StringBuffer и резервирует место для еще 16 символов без перераспределения. Объект StringBuffer выделяет место для 16 дополнительных символов, когда не запрашивается конкретная длина буфера, поскольку перераспределение — затратный процесс с точки зрения времени. Кроме того, частые перераспределения могут привести к фрагментации памяти. За счет выделения места для нескольких дополнительных символов объект StringBuffer уменьшает количество выполняемых перераспределений. Четвертая версия конструктора создает объект, содержащий последовательность символов из chars, и резервирует место для еще 16 символов.

length() и capacity()

Текущую длину StringBuffer можно узнать посредством метода length(), а общую выделенную емкость — с помощью метода capacity(). Ниже показаны их общие формы:

```
int length()
int capacity()
```

Вот пример:

```
// Работа с длиной и емкостью StringBuffer.
class StringBufferDemo {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("Hello");
    System.out.println("Буфер = " + sb);
    System.out.println("Длина = " + sb.length());
    System.out.println("Емкость = " + sb.capacity());
  }
}
```

В приведенном далее выводе программы видно, что StringBuffer резервирует добавочное пространство для дополнительных манипуляций:

```
Буфер = Hello
Длина = 5
Емкость = 21
```

Поскольку при создании объект sb инициализируется строкой "Hello", его длина равна 5, но емкость составляет 21 по причине автоматического добавления места для 16 дополнительных символов.

ensureCapacity()

При желании предварительно выделить место для определенного количества символов после конструирования объекта StringBuffer можете установить размер буфера с помощью метода sureCapacity(). Поступать так полезно, если вы заранее знаете, что будете добавлять большое количество небольших строк в StringBuffer. Метод sureCapacity() имеет следующую общую форму:

```
void ensureCapacity(int minCapacity)
```

В minCapacity указывается минимальный размер буфера. (Из соображений эффективности можно размещать и буфер с размером, превышающим minCapacity.)

setLength()

Для установки длины строки в объекте StringBuffer используйте метод setLength() с такой общей формой:

```
void setLength(int len)
```

В len задается длина строки, которая должна быть неотрицательной.

При увеличении размера строки в конец добавляются пустые символы. В случае вызова setLength() со значением меньше текущего значения, возвращаемого методом length(), символы, которые оказались за пределами новой длины, будут утрачены. В следующем разделе приводится пример программы setCharAtDemo, где метод setLength() применяется для укорачивания объекта StringBuffer.

charAt() и setCharAt()

С использованием метода charAt() можно получать значение одиночного символа из объекта StringBuffer, а с применением метода setCharAt() можно устанавливать значение символа в StringBuffer. Вот их общие формы:

```
char charAt(int where)
void setCharAt(int where, char ch)
```

В параметре where для метода charAt() указывается индекс получаемого символа. В параметре where для setCharAt() задается индекс устанавлива-

емого символа, а в ch — новое значение этого символа. Для обоих методов значение where должно быть неотрицательным и не указывать на позицию за концом строки.

Использование методов charAt() и setCharAt() демонстрируется в следующем примере:

```
// Демонстрация работы методов charAt() и setCharAt().
class setCharAtDemo {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("Hello");
    System.out.println("Буфер до = " + sb);
    System.out.println("charAt(1) до = " + sb.charAt(1));

    sb.setCharAt(1, 'i');
    sb.setLength(2);
    System.out.println("Буфер после = " + sb);
    System.out.println("charAt(1) после = " + sb.charAt(1));
  }
}
```

Ниже показан вывод, генерируемый программой:

```
Буфер до = Hello
charAt(1) до = e
Буфер после = Hi
charAt(1) после = i
```

getChars()

Для копирования подстроки из объекта StringBuffer в массив применяйте метод getChars(), имеющий такую общую форму:

```
void getChars(int sourceStart, int sourceEnd,
              char[] target, int targetStart)
```

В sourceStart передается индекс начала подстроки, а в sourceEnd — индекс, находящийся сразу после конца желаемой подстроки. Таким образом, подстрока содержит символы от sourceStart до sourceEnd-1. В target указывается массив, который получит нужные символы. Индекс внутри массива target, по которому будет копироваться подстрока, задается в targetStart. Вы должны позаботиться о том, чтобы массив target имел достаточный размер, который позволит ему уместить количество символов в указанной подстроке.

append()

Метод append() добавляет строковое представление любого другого типа данных в конец объекта StringBuffer, для которого вызывается, и имеет несколько перегруженных версий. Вот несколько его форм:

```
StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj)
```

Сначала получается строковое представление каждого параметра, после чего результат добавляется к текущему объекту StringBuffer. Каждая версия append() возвращает буфер, позволяя связывать вместе последующие вызовы, как иллюстрируется в следующем примере:

```
// Демонстрация использования append().
class appendDemo {
  public static void main(String[] args) {
    String s;
    int a = 42;
    StringBuffer sb = new StringBuffer(40);

    s = sb.append("a = ").append(a).append("!").toString();
    System.out.println(s);
  }
}
```

В результате будет получен такой вывод:

```
a = 42!
```

insert()

Метод insert() вставляет одну строку в другую. Он перегружен, чтобы принимать значения всех примитивных типов, а также экземпляры String, Object и CharSequence. Подобно append() он получает строковое представление значения, с которым вызывается, и затем эта строка вставляется в вызывающий объект StringBuffer. Ниже показано несколько его форм:

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

В index указывается индекс, по которому строка будет вставлена в вызывающий объект StringBuffer. В следующем примере между "Мне" и "Java" вставляется строка "нравится ":

```
// Демонстрация использования insert().
class insertDemo {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("Мне Java!");

    sb.insert(4, "нравится ");
    System.out.println(sb);
  }
}
```

Вот вывод, генерируемый программой:

```
Мне нравится Java!
```

reverse()

С использованием метода reverse() можно изменять на противоположный порядок следования символов в объекте StringBuffer:

```
StringBuffer reverse()
```

Метод reverse() возвращает объект, на котором был вызван, но с противоположным порядком следования символов. Его применение демонстрируется ниже в программе:

```
// Использование reverse() для изменения на противоположный
// порядка следования символов в StringBuffer.
class ReverseDemo {
  public static void main(String[] args) {
    StringBuffer s = new StringBuffer("abcdef");

    System.out.println(s);
    s.reverse();
    System.out.println(s);
  }
}
```

Программа сгенерирует такой вывод:

```
abcdef
fedcba
```

delete() и deleteCharAt()

С помощью методов delete() и deleteCharAt() можно удалять символы внутри объекта StringBuffer:

```
StringBuffer delete(int startIndex, int endIndex)
StringBuffer deleteCharAt(int loc)
```

Метод delete() удаляет последовательность символов из вызывающего объекта. В startIndex задается индекс первого удаляемого символа, а в endIndex указывается индекс, находящийся сразу за последним удаляемым символом. Таким образом, удаляемая подстрока простирается от startIndex до endIndex-1. Метод возвращает результирующий объект StringBuffer.

Метод deleteCharAt() удаляет символ по индексу, заданному в loc, и возвращает результирующий объект StringBuffer.

В следующей программе иллюстрируется использование методов delete() и deleteCharAt():

```
// Демонстрация работы delete() и deleteCharAt().
class deleteDemo {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("Строка для тестирования.");

    sb.delete(6, 10);
    System.out.println("После delete(): " + sb);

    sb.deleteCharAt(0);
    System.out.println("После deleteCharAt(): " + sb);
  }
}
```

Вот результирующий вывод:

```
После delete(): Строка тестирования.
После deleteCharAt(): трока для тестирования.
```

replace()

С применением метода replace() можно заменить один набор символов внутри объекта StringBuffer другим набором:

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

Заменяемая подстрока указывается посредством индексов startIndex и endIndex, что приводит к замене подстроки от startIndex до endIndex-1. Строка замены передается в str. Результирующий объект StringBuffer возвращается.

Ниже демонстрируется работа функции replace():

```
// Демонстрация использования replace().
class replaceDemo {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("Строка для тестирования.");

    sb.replace(7, 10, "для целей");
    System.out.println("После replace(): " + sb);
  }
}
```

Вывод выглядит так:

```
После replace(): Строка для целей тестирования.
```

substring()

Вызвав метод substring(), можно получить часть объекта StringBuffer. Вот две формы substring():

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

Первая форма возвращает подстроку, которая начинается с позиции startIndex и продолжается до конца вызывающего объекта StringBuffer. Вторая форма возвращает подстроку, начинающуюся с позиции startIndex и заканчивающуюся в позиции endIndex-1. Методы работают точно так же, как описанные ранее одноименные методы, определенные для String.

Дополнительные методы класса StringBuffer

Помимо рассмотренных выше методов класс StringBuffer предоставляет и другие. Некоторые из них приведены в табл. 18.3.

Таблица 18.3. Некоторые дополнительные методы класса `StringBuffer`

Метод	Описание
`StringBuffer appendCodePoint(int ch)`	Добавляет кодовую точку Unicode в конец вызывающего объекта. Возвращает ссылку на объект
`int codePointAt(int i)`	Возвращает кодовую точку Unicode в позиции, указанной в `i`
`int codePointBefore(int i)`	Возвращает кодовую точку Unicode в позиции, которая находится перед той, что указана в `i`
`int codePointCount(int start, int end)`	Возвращает количество кодовых точек в части вызывающего объекта, расположенной между `start` и `end-1`
`int indexOf(String str)`	Ищет в вызывающем объекте `StringBuffer` первое вхождение `str`. Возвращает индекс обнаруженного совпадения или `-1`, если совпадение не найдено
`int indexOf(String str, int startIndex)`	Ищет в вызывающем объекте `StringBuffer` первое вхождение `str`, начиная с позиции `startIndex`. Возвращает индекс обнаруженного совпадения или `-1`, если совпадение не найдено
`int lastIndexOf(String str)`	Ищет в вызывающем объекте `StringBuffer` последнее вхождение `str`. Возвращает индекс обнаруженного совпадения или `-1`, если совпадение не найдено
`int lastIndexOf(String str, int startIndex)`	Ищет в вызывающем объекте `StringBuffer` последнее вхождение `str`, начиная с позиции `startIndex`. Возвращает индекс обнаруженного совпадения или `-1`, если совпадение не найдено
`int offsetByCodePoints(int start, int num)`	Возвращает индекс внутри вызывающего объекта, который отстоит на `num` кодовых точек после начального индекса, указанного в `start`
`CharSequence subSequence(int startIndex, int stopIndex)`	Возвращает подстроку вызывающего объекта, начиная с позиции `startIndex` и заканчивая позицией `stopIndex`. Этот метод требуется для интерфейса `CharSequence`, который класс `StringBuffer` реализует
`void trimToSize()`	Требует уменьшения размера буфера символов для вызывающего объекта с целью лучшего соответствия текущему содержимому

Использование методов `indexOf()` и `lastIndexOf()` демонстрируется в следующей программе:

```
class IndexOfDemo {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("one two one");
    int i;

    i = sb.indexOf("one");
    System.out.println("Индекс первого вхождения: " + i);

    i = sb.lastIndexOf("one");
    System.out.println("Индекс последнего вхождения: " + i);
  }
}
```

Вот вывод:

```
Индекс первого вхождения: 0
Индекс последнего вхождения: 8
```

Класс StringBuilder

Класс `StringBuilder` похож на `StringBuffer` за исключением одного важного отличия: он не синхронизирован, а значит, не безопасен в отношении потоков. Преимуществом `StringBuilder` является более высокая производительность. Однако в ситуациях, когда к изменяемой строке будут обращаться несколько потоков, но не задействована внешняя синхронизация, вы должны применять `StringBuffer`, а не `StringBuilder`.

ГЛАВА

19 Исследование пакета java.lang

В настоящей главе обсуждаются классы и интерфейсы, определенные в java.lang. Как вам уже известно, пакет java.lang автоматически импортируется во все программы. Он содержит классы и интерфейсы, лежащие в основе практически всего кода Java. Это наиболее широко используемый пакет Java. Начиная с версии JDK 9, весь пакет java.lang входит в состав модуля java.base. Пакет java.lang включает в себя следующие классы:

Boolean	Float	ProcessBuilder	StringBuffer
Byte	InheritableThreadLocal	ProcessBuilder.Redirect	StringBuilder
Character	Integer	Record	System
Character.Subset	Long	Runtime	System.LoggerFinder
Character.UnicodeBlock	Math	RuntimePermission	Thread
Class	Module	Runtime.Version	ThreadGroup
ClassLoader	ModuleLayer	SecurityManager	ThreadLocal
ClassValue	ModuleLayer.Controller	Short	Throwable
Compiler	Number	StackTraceElement	Void
Double	Object	StackWalker	
Enum	Package	StrictMath	
Enum.EnumDesc	Process	String	

Кроме того, в java.lang определены следующие интерфейсы:

Appendable	Iterable	StackWalker.StackFrame
AutoCloseable	ProcessHandle	System.Logger
CharSequence	ProcessHandle.Info	Thread.UncaughtExceptionHandler
Cloneable	Readable	
Comparable	Runnable	

Определенные классы в java.lang содержат устаревшие методы, многие из которых восходят еще к Java 1.0. Устаревшие методы по-прежнему предоставляются Java для поддержки унаследованного кода, но применять их в новом коде не рекомендуется. По указанной причине в большинстве случаев устаревшие методы здесь не обсуждаются.

Оболочки примитивных типов

Как упоминалось в части I книги, примитивные типы вроде `int` и `char` используются в Java из соображений производительности. Такие типы данных не являются частью иерархии объектов. Они передаются в методы по значению и не могут быть переданы напрямую по ссылке. Кроме того, два метода не могут ссылаться на *один и тот же экземпляр* `int`. Иногда у вас будет возникать необходимость создавать объектное представление для одного из примитивных типов. Например, в главе 20 обсуждаются классы коллекций, которые имеют дело только с объектами; чтобы сохранить примитивный тип в одном из таких классов, примитивный тип понадобится поместить в класс. Для удовлетворения этой потребности в Java предусмотрены классы, соответствующие каждому примитивному типу. Такие классы по существу инкапсулируют (или *содержат*) внутри себя примитивные типы, поэтому их обычно называют *оболочками типов*. Оболочки типов были представлены в главе 12, а здесь они рассматриваются более подробно.

Прежде чем начать, необходимо отметить важный момент. Начиная с версии JDK 16, классы оболочек примитивных типов документируются как *основанные на значениях*. В итоге к ним применяются различные правила и ограничения. Скажем, вы должны избегать использования экземпляров класса на основе значений для синхронизации. Дополнительную информацию о классах, основанных на значениях, ищите в главе 13.

Number

Абстрактный класс `Number` является суперклассом, реализуемым классами оболочек числовых типов `byte`, `short`, `int`, `long`, `float` и `double`. Класс `Number` имеет абстрактные методы, которые возвращают значение объекта в различных числовых форматах. Например, метод `doubleValue()` возвращает значение типа `double`, метод `floatValue()` — значение типа `float` и т.д. Такие методы перечислены ниже:

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

Возвращаемые этими методами значения могут быть округлены, усечены или становиться "мусорными" из-за результатов сужающего преобразования.

Для класса `Number` определены конкретные подклассы, которые хранят явные значения всех примитивных числовых типов: `Double`, `Float`, `Byte`, `Short`, `Integer` и `Long`.

Double и Float

Классы `Double` и `Float` являются оболочками для значений с плавающей точкой типа `double` и `float` соответственно. Вот конструкторы класса `Float`:

```
Float(double num) Float(float num)
Float(String str) throws NumberFormatException
```

Объекты Float могут конструироваться со значениями типа float или double. Их также можно создавать из строкового представления числа с плавающей точкой. Начиная с версии JDK 9, эти конструкторы объявлены нерекомендуемыми, а в версии JDK 16 они помечены как устаревшие и подлежащие удалению. Настоятельно рекомендуемой альтернативой считается метод valueOf().

Ниже показаны конструкторы класса Double:

```
Double(double num)
Double(String str) throws NumberFormatException
```

Объекты Double могут конструироваться со значением типа double или строкой, содержащей значение с плавающей точкой. Начиная с версии JDK 9, такие конструкторы объявлены нерекомендуемыми, а в версии JDK 16 они помечены как устаревшие и подлежащие удалению. Настоятельно рекомендуемой альтернативой является метод valueOf().

В табл. 19.1 кратко описаны распространенные методы, определенные в классе Float, а в табл. 19.2 — часто применяемые методы, которые определены в Double. Начиная с версии JDK 12, классы Float и Double также реализуют интерфейсы Constable и ConstantDesc. В классах Float и Double определены следующие константы:

BYTES	Ширина в байтах значения типа float или double
MAX_EXPONENT	Максимальный показатель степени
MAX_VALUE	Максимальное положительное значение
MIN_EXPONENT	Минимальный показатель степени
MIN_NORMAL	Минимальное положительное нормальное значение
MIN_VALUE	Минимальное положительное значение
NaN	Не число
POSITIVE_INFINITY	Положительная бесконечность
NEGATIVE_INFINITY	Отрицательная бесконечность
SIZE	Ширина в битах значения внутри оболочки
TYPE	Объект Class для значения типа float или double

В приведенном далее примере создаются два объекта Double — один с использованием значения double и еще один за счет передачи строки, которая может быть разобрана как значение double:

```
class DoubleDemo {
  public static void main(String[] args) {
    Double d1 = Double.valueOf(3.14159);
    Double d2 = Double.valueOf("314159E-5");

    System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
  }
}
```

Таблица 19.1. Часто используемые методы класса `Float`

Метод	Описание
`byte byteValue()`	Возвращает значение вызывающего объекта как число типа `byte`

`static int compare(`
`float num1,`
`float num2)`
	Сравнивает значения `num1` и `num2`. Возвращает `0`, если значения равны. Возвращает отрицательное значение, если `num1` меньше `num2`. Возвращает положительное значение, если `num1` больше `num2`

`int compareTo(Float f)`	Сравнивает числовое значение вызывающего объекта со значением `f`. Возвращает `0`, если значения равны. Возвращает отрицательное значение, если вызывающий объект содержит меньшее значение. Возвращает положительное значение, если вызывающий объект содержит большее значение
`double doubleValue()`	Возвращает значение вызывающего объекта как число типа `double`
`boolean equals(` `Object FloatObj)`	Возвращает `true`, если вызывающий объект `Float` эквивалентен `FloatObj`, или `false` в противном случае
`static int` `floatToIntBits(float num)`	Возвращает совместимый с IEEE битовый шаблон одинарной точности, который соответствует `num`

`static int`
`floatToRawIntBits(`
`float num)`
	Возвращает совместимый с IEEE битовый шаблон одинарной точности, который соответствует `num`. Значение `NaN` предохраняется

`float floatValue()`	Возвращает значение вызывающего объекта как число типа `float`
`int hashCode()`	Возвращает хеш-код для вызывающего объекта
`static int hashCode(` `float num)`	Возвращает хеш-код для `num`
`static float` `intBitsToFloat(int num)`	Возвращает эквивалент типа `float` совместимого с IEEE битового шаблона одинарной точности, указанного в `num`
`int intValue()`	Возвращает значение вызывающего объекта как число типа `int`
`static boolean` `isFinite(float num)`	Возвращает `true`, если значение `num` — не `NaN` и не бесконечность

Окончание табл. 19.1

Метод	Описание
`boolean isInfinite()`	Возвращает `true`, если вызывающий объект содержит бесконечное значение, или `false` в противном случае
`static boolean isInfinite(float num)`	Возвращает `true`, если в `num` указано бесконечное значение, или `false` в противном случае
`boolean isNaN()`	Возвращает `true`, если вызывающий объект содержит значение, не являющееся числом, или `false` в противном случае
`static boolean isNaN( float num)`	Возвращает `true`, если в `num` указано значение, не являющееся числом, или `false` в противном случае
`long longValue()`	Возвращает значение вызывающего объекта как число типа `long`
`static float max(float val, float val2)`	Возвращает большее значение среди `val` и `val2`
`static float min(float val, float val2)`	Возвращает меньшее значение среди `val` и `val2`
`static float parseFloat( String str) throws NumberFormatException`	Возвращает эквивалент типа `float` числа, содержащегося в строке, которая указана в `str` с использованием основания 10
`short shortValue()`	Возвращает значение вызывающего объекта как число типа `short`
`static float sum(float val, float val2)`	Возвращает результат сложения `val` и `val2`
`static String toHexString(float num)`	Возвращает строку, содержащую значение `num` в шестнадцатеричном формате
`String toString()`	Возвращает строковый эквивалент вызывающего объекта
`static String toString(float num)`	Возвращает строковый эквивалент значения, указанного в `num`
`static Float valueOf( float num)`	Возвращает объект `Float`, который содержит значение, переданное в `num`
`static Float valueOf(String str) throws NumberFormatException`	Возвращает объект `Float`, который содержит значение, заданное строкой в `str`

Таблица 19.2. Часто используемые методы класса Double

Метод	Описание
`byte byteValue()`	Возвращает значение вызывающего объекта как число типа byte
`static int compare( double num1, double num2)`	Сравнивает значения num1 и num2. Возвращает 0, если значения равны. Возвращает отрицательное значение, если num1 меньше num2. Возвращает положительное значение, если num1 больше num2
`int compareTo(Double d)`	Сравнивает числовое значение вызывающего объекта со значением d. Возвращает 0, если значения равны. Возвращает отрицательное значение, если вызывающий объект содержит меньшее значение. Возвращает положительное значение, если вызывающий объект содержит большее значение
`static long doubleToLongBits( double num)`	Возвращает совместимый с IEEE битовый шаблон двойной точности, который соответствует num
`static long doubleToRawLongBits( double num)`	Возвращает совместимый с IEEE битовый шаблон двойной точности, который соответствует num. Значение NaN предохраняется
`double doubleValue()`	Возвращает значение вызывающего объекта как число типа double
`boolean equals( Object DoubleObj)`	Возвращает true, если вызывающий объект Double эквивалентен DoubleObj, или false в противном случае
`float floatValue()`	Возвращает значение вызывающего объекта как число типа float
`int hashcode()`	Возвращает хеш-код для вызывающего объекта
`static int hashCode( double num)`	Возвращает хеш-код для num
`int intValue()`	Возвращает значение вызывающего объекта как число типа int
`static boolean isFinite( double num)`	Возвращает true, если значение num — не NaN и не бесконечность
`boolean isInfinite()`	Возвращает true, если вызывающий объект содержит бесконечное значение, или false в противном случае

Окончание табл. 19.2

Метод	Описание
`static boolean isInfinite(double num)`	Возвращает `true`, если в `num` указано бесконечное значение, или `false` в противном случае
`boolean isNaN()`	Возвращает `true`, если вызывающий объект содержит значение, не являющееся числом, или `false` в противном случае
`static boolean isNaN(double num)`	Возвращает `true`, если в `num` указано значение, не являющееся числом, или `false` в противном случае
`static double longBitsToDouble(long num)`	Возвращает эквивалент типа `double` совместимого с IEEE битового шаблона двойной точности, указанного в `num`
`long longValue()`	Возвращает значение вызывающего объекта как число типа `long`
`static double max( double val, double val2)`	Возвращает большее значение среди `val` и `val2`
`static double min( double val, double val2)`	Возвращает меньшее значение среди `val` и `val2`
`static double parseDouble(String str) throws NumberFormatException`	Возвращает эквивалент типа `double` числа, содержащегося в строке, которая указана в `str` с использованием основания 10
`short shortValue()`	Возвращает значение вызывающего объекта как число типа `short`
`static double sum(double val, double val2)`	Возвращает результат сложения `val` и `val2`
`static String toHexString(double num)`	Возвращает строку, содержащую значение `num` в шестнадцатеричном формате
`String toString()`	Возвращает строковый эквивалент вызывающего объекта
`static String toString(double num)`	Возвращает строковый эквивалент значения, указанного в `num`
`static Double valueOf(double num)`	Возвращает объект `Double`, который содержит значение, переданное в `num`
`static Double valueOf(String str) throws NumberFormatException`	Возвращает объект `Double`, который содержит значение, заданное строкой в `str`

В следующем выводе видно, что обе версии valueOf() создали идентичные экземпляры Double, о чем можно судить по факту возвращения значения true методом equals():

```
3.14159 = 3.14159 -> true
```

Методы isInfinite() и isNaN()

Классы Float и Double предоставляют методы isInfinite() и isNaN(), которые помогают манипулировать двумя специальными значениями типа double и float. Эти методы проверяют на предмет равенства двум уникальным значениям: бесконечности и NaN (не числу). Метод isInfinite() возвращает true, если проверяемое значение бесконечно велико или мало по величине. Метод isNaN() возвращает true, если проверяемое значение не является числом.

В следующем примере создаются два объекта Double; один содержит бесконечное, а другой — нечисловое значение:

```
// Демонстрация работы isInfinite() и isNaN().
class InfNaN {
  public static void main(String[] args) {
    Double d1 = Double.valueOf(1/0.);
    Double d2 = Double.valueOf(0/0.);

    System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1.isNaN());
    System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isNaN());
  }
}
```

Вот вывод, генерируемый программой:

```
Infinity: true, false
NaN: false, true
```

Byte, Short, Integer и Long

Классы Byte, Short, Integer и Long являются оболочками для целочисленных типов byte, short, int и long соответственно. Ниже показаны их конструкторы:

```
Byte(byte num)
Byte(String str) throws NumberFormatException
Short(short num)
Short(String str) throws NumberFormatException
Integer(int num)
Integer(String str) throws NumberFormatException
Long(long num)
Long(String str) throws NumberFormatException
```

Легко заметить, что объекты могут создаваться из числовых значений или строк, содержащих допустимые целые числа. Начиная с версии JDK 9, перечисленные выше конструкторы объявлены нерекомендуемыми, а в версии JDK 16 они помечены как устаревшие и подлежащие удалению. Настоятельно рекомендуемой альтернативой является метод valueOf().

В табл. 19.3–19.6 кратко описаны распространенные методы, которые определены в этих классах. Как видите, в них определены методы, предназначенные для извлечения целых чисел из строк и преобразования строк в целые числа. Некоторые варианты методов позволяют указывать *основание системы счисления* для преобразования, например, 2 для двоичной, 8 для восьмеричной, 10 для десятичной и 16 для шестнадцатеричной. Начиная с версии JDK 12, классы `Integer` и `Long` вдобавок реализуют интерфейсы `Constable` и `ConstantDesc`. Начиная с версии JDK 15, классы `Byte` и `Short` тоже реализуют интерфейс `Constable`.

Таблица 19.3. Часто используемые методы класса Byte

Метод	Описание
`byte byteValue()`	Возвращает значение вызывающего объекта как число типа `byte`
`static int compare (byte num1, byte num2)`	Сравнивает значения `num1` и `num2`. Возвращает 0, если значения равны. Возвращает отрицательное значение, если `num1` меньше `num2`. Возвращает положительное значение, если `num1` больше `num2`
`int compareTo(Byte b)`	Сравнивает числовое значение вызывающего объекта со значением `b`. Возвращает 0, если значения равны. Возвращает отрицательное значение, если вызывающий объект содержит меньшее значение. Возвращает положительное значение, если вызывающий объект содержит большее значение
`static int compareUnsigned (byte num1, byte num2)`	Выполняет беззнаковое сравнение `num1` и `num2`. Возвращает 0, если значения равны. Возвращает отрицательное значение, если `num1` меньше `num2`. Возвращает положительное значение, если `num1` больше `num2`
`static Byte decode (String str) throws NumberFormatException`	Возвращает объект `Byte`, который содержит значение, заданное строкой в `str`
`double doubleValue()`	Возвращает значение вызывающего объекта как число типа `double`
`boolean equals (Object ByteObj)`	Возвращает `true`, если вызывающий объект `Byte` эквивалентен `ByteObj`, или `false` в противном случае
`float floatValue()`	Возвращает значение вызывающего объекта как число типа `float`

Окончание табл. 19.3

Метод	Описание
`int hashCode()`	Возвращает хеш-код для вызывающего объекта
`static int hashCode(byte num)`	Возвращает хеш-код для num
`int intValue()`	Возвращает значение вызывающего объекта как число типа int
`long longValue()`	Возвращает значение вызывающего объекта как число типа long
`static byte parseByte (String str) throws NumberFormatException`	Возвращает эквивалент типа byte числа, содержащегося в строке, которая указана в str с использованием основания 10
`static byte parseByte (String str, int radix) throws NumberFormatException`	Возвращает эквивалент типа byte числа, содержащегося в строке, которая указана в str с использованием основания radix
`short shortValue()`	Возвращает значение вызывающего объекта как число типа short
`String toString()`	Возвращает строку, содержащую десятичный эквивалент вызывающего объекта
`static String toString(byte num)`	Возвращает строку, содержащую десятичный эквивалент num
`static int toUnsignedInt(byte val)`	Возвращает значение вызывающего объекта как целое без знака
`static long toUnsignedLong(byte val)`	Возвращает значение val как длинное целое без знака
`static Byte valueOf (byte num)`	Возвращает объект Byte, который содержит значение, переданное в num
`static Byte valueOf (String str) throws NumberFormatException`	Возвращает объект Byte, который содержит значение, заданное строкой в str
`static Byte valueOf (String str, int radix) throws NumberFormatException`	Возвращает объект Byte, который содержит значение, заданное строкой в str с использованием основания radix

Таблица 19.4. Часто используемые методы класса Short

Метод	Описание
`byte byteValue()`	Возвращает значение вызывающего объекта как число типа byte
`static int compare (short num1, short num2`	Сравнивает значения num1 и num2. Возвращает 0, если значения равны. Возвращает отрицательное значение, если num1 меньше num2. Возвращает положительное значение, если num1 больше num2
`int compareTo(Short s)`	Сравнивает числовое значение вызывающего объекта со значением s. Возвращает 0, если значения равны. Возвращает отрицательное значение, если вызывающий объект содержит меньшее значение. Возвращает положительное значение, если вызывающий объект содержит большее значение
`static int compareUnsigned (short num1, short num2)`	Выполняет беззнаковое сравнение num1 и num2. Возвращает 0, если значения равны. Возвращает отрицательное значение, если num1 меньше num2. Возвращает положительное значение, если num1 больше num2
`static Short decode (String str) throws NumberFormatException`	Возвращает объект Short, который содержит значение, заданное строкой в str
`double doubleValue()`	Возвращает значение вызывающего объекта как число типа double
`boolean equals (Object ShortObj)`	Возвращает true, если вызывающий объект Short эквивалентен ShortObj, или false в противном случае
`float floatValue()`	Возвращает значение вызывающего объекта как число типа float
`int hashCode()`	Возвращает хеш-код для вызывающего объекта
`static int hashCode (short num)`	Возвращает хеш-код для num
`int intValue()`	Возвращает значение вызывающего объекта как число типа int
`long longValue()`	Возвращает значение вызывающего объекта как число типа long

Окончание табл. 19.4

Метод	Описание
`static short parseShort (String str) throws NumberFormatException`	Возвращает эквивалент типа `short` числа, содержащегося в строке, которая указана в `str` с использованием основания 10
`static short parseShort (String str, int radix) throws NumberFormatException`	Возвращает эквивалент типа `short` числа, содержащегося в строке, которая указана в `str` с использованием основания `radix`
`static short reverseBytes(short num)`	Меняет местами старший и младший байты числа и возвращает результат
`short shortValue()`	Возвращает значение вызывающего объекта как число типа `short`
`String toString()`	Возвращает строку, содержащую десятичный эквивалент вызывающего объекта
`static String toString(short num)`	Возвращает строку, содержащую десятичный эквивалент `num`
`static int toUnsignedInt(short val)`	Возвращает значение `val` как целое без знака
`static long toUnsignedLong(short val)`	Возвращает значение `val` как длинное целое без знака
`static Short valueOf(short num)`	Возвращает объект `Short`, который содержит значение, переданное в `num`
`static Short valueOf (String str) throws NumberFormatException`	Возвращает объект `Short`, который содержит значение, заданное строкой в `str` с использованием основания 10
`static Short valueOf (String str, int radix) throws NumberFormatException`	Возвращает объект `Short`, который содержит значение, заданное строкой в `str` с использованием основания `radix`

Таблица 19.5. Часто используемые методы класса `Integer`

Метод	Описание
`static int bitCount(int num)`	Возвращает количество установленных битов в num
`byte byteValue()`	Возвращает значение вызывающего объекта как число типа byte
`static int compare (int num1, int num2)`	Сравнивает значения num1 и num2. Возвращает 0, если значения равны. Возвращает отрицательное значение, если num1 меньше num2. Возвращает положительное значение, если num1 больше num2
`int compareTo(Integer i)`	Сравнивает числовое значение вызывающего объекта со значением i. Возвращает 0, если значения равны. Возвращает отрицательное значение, если вызывающий объект содержит меньшее значение. Возвращает положительное значение, если вызывающий объект содержит большее значение
`static int compareUnsigned (int num1, int num2)`	Выполняет беззнаковое сравнение num1 и num2. Возвращает 0, если значения равны. Возвращает отрицательное значение, если num1 меньше num2. Возвращает положительное значение, если num1 больше num2
`static Integer decode (String str) throws NumberFormatException`	Возвращает объект Integer, который содержит значение, заданное строкой в str
`static int divideUnsigned (int dividend, int divisor)`	Возвращает результат беззнакового деления dividend на divisor в виде значения без знака
`double doubleValue()`	Возвращает значение вызывающего объекта как число типа double
`boolean equals (Object IntegerObj)`	Возвращает true, если вызывающий объект Integer эквивалентен IntegerObj, или false в противном случае
`float floatValue()`	Возвращает значение вызывающего объекта как число типа float

Продолжение табл. 19.5

Метод	Описание
`static Integer getInteger (String propertyName)`	Возвращает значение, ассоциированное со свойством среды, которое указано в `propertyName`. В случае неудачи возвращается `null`
`static Integer getInteger (String propertyName, int default)`	Возвращает значение, ассоциированное со свойством среды, которое указано в `propertyName`. В случае неудачи возвращается значение `default`
`static Integer getInteger (String propertyName, Integer default)`	Возвращает значение, ассоциированное со свойством среды, которое указано в `propertyName`. В случае неудачи возвращается значение `default`
`int hashCode()`	Возвращает хеш-код для вызывающего объекта
`static int hashCode(int num)`	Возвращает хеш-код для `num`
`static int highestOneBit (int num)`	Определяет позицию самого старшего установленного бита в `num`. Возвращает значение, в котором установлен только этот бит. При отсутствии битов, установленных в единицу, возвращается ноль
`int intValue()`	Возвращает значение вызывающего объекта как число типа `int`
`long longValue()`	Возвращает значение вызывающего объекта как число типа `long`
`static int lowestOneBit (int num)`	Определяет позицию самого младшего установленного бита в `num`. Возвращает значение, в котором установлен только этот бит. При отсутствии битов, установленных в единицу, возвращается ноль
`static int max (int val, int val2)`	Возвращает большее значение среди `val` и `val2`
`static int min (int val, int val2)`	Возвращает меньшее значение среди `val` и `val2`
`static int numberOfLeadingZeros(int num)`	Возвращает количество старших нулевых битов, которые предшествуют первому старшему установленному биту в `num`. Если значение `num` равно нулю, тогда возвращается 32

Продолжение табл. 19.5

Метод	Описание
`static int numberOfTrailingZeros (int num)`	Возвращает количество младших нулевых битов, которые предшествуют первому младшему установленному биту в num. Если значение num равно нулю, тогда возвращается 32
`static int parseInt (CharSequence chars, int startIdx, int stopIdx, int radix) throws NumberFormatException`	Возвращает эквивалент типа int числа, которое содержится в последовательности, указанной в chars, между индексами startIdx и stopIdx-1, с использованием основания radix
`static int parseInt (String str) throws NumberFormatException`	Возвращает эквивалент типа int числа, содержащегося в строке, которая указана в str с использованием основания 10
`static int parseInt (String str, int radix) throws NumberFormatException`	Возвращает эквивалент типа int числа, содержащегося в строке, которая указана в str с использованием основания radix
`static int parseUnsignedInt (CharSequence chars, int startIdx, int stopIdx, int radix) throws NumberFormatException`	Возвращает эквивалент типа int числа без знака, которое содержится в последовательности, указанной в chars, между индексами startIdx и stopIdx-1, с использованием основания radix
`static int parseUnsignedInt (String str) throws NumberFormatException`	Возвращает эквивалент типа int числа без знака, содержащегося в строке, которая указана в str с использованием основания 10
`static int parseUnsignedInt (String str, int radix) throws NumberFormatException`	Возвращает эквивалент типа int числа без знака, содержащегося в строке, которая указана в str с использованием основания radix
`static int remainderUnsigned (int dividend, int divisor)`	Возвращает остаток от беззнакового деления dividend на divisor в виде значения без знака
`static int reverse(int num)`	Изменяет порядок следования битов в num на противоположный и возвращает результат

Продолжение табл. 19.5

Метод	Описание
`static int reverseBytes (int num)`	Изменяет порядок следования байтов в num на противоположный и возвращает результат
`static int rotateLeft (int num, int n)`	Возвращает результат циклического сдвига num влево на n позиций
`static int rotateRight (int num, int n)`	Возвращает результат циклического сдвига num вправо на n позиций
`short shortValue()`	Возвращает значение вызывающего объекта как число типа short
`static int signum(int num)`	Возвращает -1, если значение num является отрицательным, 0 — если нулевым, и 1 — если положительным
`static int sum(int val, int val2)`	Возвращает результат сложения val и val2
`static String toBinaryString(int num)`	Возвращает строку, содержащую двоичный эквивалент num
`static String toHexString (int num)`	Возвращает строку, содержащую шестнадцатеричный эквивалент num
`static String toOctalString(int num)`	Возвращает строку, содержащую восьмеричный эквивалент num
`String toString()`	Возвращает строку, содержащую десятичный эквивалент вызывающего объекта
`static String toString (int num)`	Возвращает строку, содержащую десятичный эквивалент num
`static String toString (int num, int radix)`	Возвращает строку, содержащую десятичный эквивалент num с использованием основания radix
`static long toUnsignedLong(int val)`	Возвращает значение val как длинное целое без знака
`static String toUnsignedString(int val)`	Возвращает строку, содержащую десятичное значение val как целое без знака
`static String toUnsignedString(int val, int radix)`	Возвращает строку, содержащую десятичное значение val как целое без знака с использованием основания radix
`static Integer valueOf (int num)`	Возвращает объект Integer, который содержит значение, переданное в num

Окончание табл. 19.5

Метод	Описание
`static Integer valueOf (String str) throws NumberFormatException`	Возвращает объект `Integer`, который содержит значение, заданное строкой в `str`
`static Integer valueOf (String str, int radix) throws NumberFormatException`	Возвращает объект `Integer`, который содержит значение, заданное строкой в `str` с использованием основания `radix`

Таблица 19.6. Часто используемые методы класса Long

Метод	Описание
`static int bitCount (ong num)`	Возвращает количество установленных битов в `num`
`byte byteValue()`	Возвращает значение вызывающего объекта как число типа `byte`
`static int compare (long num1, long num2)`	Сравнивает значения `num1` и `num2`. Возвращает 0, если значения равны. Возвращает отрицательное значение, если `num1` меньше `num2`. Возвращает положительное значение, если `num1` больше `num2`
`int compareTo(Long l)`	Сравнивает числовое значение вызывающего объекта со значением `l`. Возвращает 0, если значения равны. Возвращает отрицательное значение, если вызывающий объект содержит меньшее значение. Возвращает положительное значение, если вызывающий объект содержит большее значение
`static int compareUnsigned (long num1, long num2)`	Выполняет беззнаковое сравнение `num1` и `num2`. Возвращает 0, если значения равны. Возвращает отрицательное значение, если `num1` меньше `num2`. Возвращает положительное значение, если `num1` больше `num2`
`static Long decode (String str) throws NumberFormatException`	Возвращает объект `Long`, который содержит значение, заданное строкой в `str`
`static long divideUnsigned (long dividend, long divisor)`	Возвращает результат беззнакового деления `dividend` на `divisor` в виде значения без знака

Продолжение табл. 19.6

Метод	Описание
`double doubleValue()`	Возвращает значение вызывающего объекта как число типа `double`
`boolean equals (Object LongObj)`	Возвращает `true`, если вызывающий объект `Long` эквивалентен `LongObj`, или `false` в противном случае
`float floatValue()`	Возвращает значение вызывающего объекта как число типа `float`
`static Long getLong (String propertyName)`	Возвращает значение, ассоциированное со свойством среды, которое указано в `propertyName`. В случае неудачи возвращается `null`
`static Long getLong (String propertyName, long default)`	Возвращает значение, ассоциированное со свойством среды, которое указано в `propertyName`. В случае неудачи возвращается значение `default`
`static Long getLong (String propertyName, Long default)`	Возвращает значение, ассоциированное со свойством среды, которое указано в `propertyName`. В случае неудачи возвращается значение `default`
`int hashCode()`	Возвращает хеш-код для вызывающего объекта
`static int hashCode (long num)`	Возвращает хеш-код для `num`
`static long highestOneBit (long num)`	Определяет позицию самого старшего установленного бита в `num`. Возвращает значение, в котором установлен только этот бит. При отсутствии битов, установленных в единицу, возвращается ноль
`int intValue()`	Возвращает значение вызывающего объекта как число типа `int`
`long longValue()`	Возвращает значение вызывающего объекта как число типа `long`
`static long lowestOneBit (long num)`	Определяет позицию самого младшего установленного бита в `num`. Возвращает значение, в котором установлен только этот бит. При отсутствии битов, установленных в единицу, возвращается ноль

Продолжение табл. 19.6

Метод	Описание
`static long max (long val, long val2)`	Возвращает большее значение среди val и val2
`static long min (long val, long val2)`	Возвращает меньшее значение среди val и val2
`static int numberOfLeadingZeros (long num)`	Возвращает количество старших нулевых битов, которые предшествуют первому старшему установленному биту в num. Если значение num равно нулю, тогда возвращается 64
`static int numberOfTrailingZeros (long num)`	Возвращает количество младших нулевых битов, которые предшествуют первому младшему установленному биту в num. Если значение num равно нулю, тогда возвращается 64
`static long parseLong (CharSequence chars, int startIdx, int stopIdx, int radix) throws NumberFormatException`	Возвращает эквивалент типа long числа, которое содержится в последовательности, указанной в chars, между индексами startIdx и stopIdx-1, с использованием основания radix
`static long parseLong(String str) throws NumberFormatException`	Возвращает эквивалент типа long числа, содержащегося в строке, которая указана в str с использованием основания 10
`static long parseLong (String str, int radix) throws NumberFormatException`	Возвращает эквивалент типа long числа, содержащегося в строке, которая указана в str с использованием основания radix
`static long parseUnsignedLong (CharSequence chars, int startIdx, int stopIdx, int radix) throws NumberFormatException`	Возвращает эквивалент типа long числа без знака, которое содержится в последовательности, указанной в chars, между индексами startIdx и stopIdx-1, с использованием основания radix
`static long parseUnsignedLong(String str) throws NumberFormatException`	Возвращает эквивалент типа long числа без знака, содержащегося в строке, которая указана в str с использованием основания 10

Продолжение табл. 19.6

Метод	Описание
`static long parseUnsignedLong (String str, int radix) throws NumberFormatException`	Возвращает эквивалент типа `long` числа без знака, содержащегося в строке, которая указана в `str` с использованием основания `radix`
`static long remainderUnsigned (long dividend, long divisor)`	Возвращает остаток от беззнакового деления `dividend` на `divisor` в виде значения без знака
`static long reverse (long num)`	Изменяет порядок следования битов в `num` на противоположный и возвращает результат
`static long reverseBytes (long num)`	Изменяет порядок следования байтов в `num` на противоположный и возвращает результат
`static long rotateLeft (long num, int n)`	Возвращает результат циклического сдвига `num` влево на `n` позиций
`static long rotateRight (long num, int n)`	Возвращает результат циклического сдвига `num` вправо на `n` позиций
`short shortValue()`	Возвращает значение вызывающего объекта как число типа `short`
`static int signum(long num)`	Возвращает `-1`, если значение `num` является отрицательным, `0` — если нулевым, и `1` — если положительным
`static long sum (long val, long val2)`	Возвращает результат сложения `val` и `val2`
`static String toBinaryString (long num)`	Возвращает строку, содержащую двоичный эквивалент `num`
`static String toHexString(long num)`	Возвращает строку, содержащую шестнадцатеричный эквивалент `num`
`static String toOctalString(long num)`	Возвращает строку, содержащую восьмеричный эквивалент `num`
`String toString()`	Возвращает строку, содержащую десятичный эквивалент вызывающего объекта
`static String toString (long num)`	Возвращает строку, содержащую десятичный эквивалент `num`
`static String toString (long num, int radix)`	Возвращает строку, содержащую десятичный эквивалент `num` с использованием основания `radix`

Окончание табл. 19.6

Метод	Описание
`static String toUnsignedString(long val)`	Возвращает строку, содержащую десятичное значение `val` как целое без знака
`static String toUnsignedString (long val, int radix)`	Возвращает строку, содержащую значение `val` как целое без знака с использованием основания `radix`
`static Long valueOf (long num)`	Возвращает объект `Long`, который содержит значение, переданное в `num`
`static Long valueOf (String str) throws NumberFormatException`	Возвращает объект `Long`, который содержит значение, заданное строкой в `str`
`static Long valueOf(String str, int radix) throws NumberFormatException`	Возвращает объект `Long`, который содержит значение, заданное строкой в `str` с использованием основания `radix`

Определены следующие константы:

`BYTES`	Ширина в байтах значения целочисленного типа
`MIN_VALUE`	Минимальное значение
`MAX_VALUE`	Максимальное значение
`SIZE`	Ширина в битах значения внутри оболочки
`TYPE`	Объект `Class` для значения типа `byte`, `short`, `int` или `long`

Преобразование чисел в строки и обратно

Одной из наиболее распространенных задач программирования является преобразование строкового представления числа в его внутренний двоичный формат. К счастью, в Java предлагается простой способ делать это. Классы `Byte`, `Short`, `Integer` и `Long` предоставляют методы `parseByte()`, `parseShort()`, `parseInt()` и `parseLong()`. Указанные методы возвращают эквивалент типа `byte`, `short`, `int` или `long` числовой строки, с которой они вызываются. (Аналогичные методы также существуют для классов `Float` и `Double`.) В показанной ниже программе демонстрируется работа метода `parseInt()`. В ней суммируется список целых чисел, введенных пользователем. Целые числа читаются с помощью `readLine()` и посредством `parseInt()` строки, содержащие прочитанные числа, преобразуются в эквиваленты типа `int`.

```
/* В программе суммируется список целых чисел, введенных пользователем.
   С применением parseInt() строковое представление каждого числа
   преобразуется в тип int.
*/
```

```
import java.io.*;

class ParseDemo {
  public static void main(String[] args) throws IOException
  {
    // Создать объект BufferedReader с использованием System.in.
    BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in, System.console().charset()));
    String str;
    int i;
    int sum=0;

    System.out.println("Вводите числа или 0 для выхода.");
    do {
      str = br.readLine();
      try {
        i = Integer.parseInt(str);
      } catch(NumberFormatException e) {
        System.out.println("Недопустимый формат");
        i = 0;
      }
      sum += i;
      System.out.println("Текущая сумма: " + sum);
    } while(i != 0);
  }
}
```

Для преобразования целого числа в десятичную строку применяются версии `toString()`, определенные в классах `Byte`, `Short`, `Integer` или `Long`. Классы `Integer` и `Long` также предлагают методы `toBinaryString()`, `toHexString()` и `toOctalString()`, которые преобразуют значение в двоичную, шестнадцатеричную или восьмеричную строку соответственно.

В следующей программе демонстрируется двоичное, шестнадцатеричное и восьмеричное преобразование:

```
/* Преобразование целого числа в двоичную,
   шестнадцатеричную и восьмеричную строку.
*/

class StringConversions {
  public static void main(String[] args) {
    int num = 19648;
    System.out.println(num + " в двоичной форме: " +
                       Integer.toBinaryString(num));
    System.out.println(num + " в восьмеричной форме: " +
                       Integer.toOctalString(num));
    System.out.println(num + " в шестнадцатеричной форме: " +
                       Integer.toHexString(num));
  }
}
```

Вот вывод, генерируемый программой:

```
19648 в двоичной форме: 100110011000000
19648 в восьмеричной форме: 46300
19648 в шестнадцатеричной форме: 4cc0
```

Character

Класс Character является простой оболочкой для char. Ниже показан конструктор класса Character:

```
Character(char ch)
```

В ch указывается символ, который будет помещен внутрь создаваемого объекта Character. Начиная с версии JDK 9, этот конструктор помечен как нерекомендуемый, а в версии JDK 16 он объявлен устаревшим и подлежащим удалению. Настоятельно рекомендуемой альтернативой считается метод valueOf().

Чтобы получить значение char, содержащееся в объекте Character, нужно вызвать метод charValue():

```
char charValue()
```

Он возвращает символ.

В классе Character определено несколько констант, в том числе:

BYTES	Ширина в байтах значения типа char
MAX_RADIX	Наибольшее основание системы счисления
MIN_RADIX	Наименьшее основание системы счисления
MAX_VALUE	Наибольшее символьное значение
MIN_VALUE	Наименьшее символьное значение
TYPE	Объект Class для char

В состав класса Character входят несколько статических методов, которые классифицируют символы и изменяют их регистр. Подборка таких методов приведена в табл. 19.7. В следующем примере демонстрируется работа некоторых статических методов:

```
// Демонстрация работы ряда методов isX() класса Character.
class IsDemo {
  public static void main(String[] args) {
    char[] a = {'a', 'b', '5', '?', 'A', ' '};

    for(int i=0; i<a.length; i++) {
      if(Character.isDigit(a[i]))
        System.out.println(a[i] + " - цифра.");
      if(Character.isLetter(a[i]))
        System.out.println(a[i] + " - буква.");
      if(Character.isWhitespace(a[i]))
        System.out.println(a[i] + " - пробельный символ.");
      if(Character.isUpperCase(a[i]))
        System.out.println(a[i] + " - буква в верхнем регистре.");
      if(Character.isLowerCase(a[i]))
        System.out.println(a[i] + " - буква в нижнем регистре.");
    }
  }
}
```

Ниже показан вывод:

```
a - буква.
a - буква в нижнем регистре.
b - буква.
b - буква в нижнем регистре.
5 - цифра.
A - буква.
A - буква в верхнем регистре.
  - пробельный символ.
```

Таблица 19.7. Часто используемые методы класса Character

Метод	Описание
`static boolean isDefined (char ch)`	Возвращает `true`, если символ `ch` определен в Unicode, или `false` в противном случае
`static boolean isDigit (char ch)`	Возвращает `true`, если `ch` является цифрой, или `false` в противном случае
`static boolean isIdentifierIgnorable (char ch)`	Возвращает `true`, если `ch` должен игнорироваться в идентификаторе, или `false` в противном случае
`static boolean isISOControl (char ch)`	Возвращает `true`, если `ch` является управляющим символом ISO, или `false` в противном случае
`static boolean isJavaIdentifierPart (char ch)`	Возвращает `true`, если `ch` разрешено быть частью идентификатора Java (кроме первого символа), или `false` в противном случае
`static boolean isJavaIdentifierStart (char ch)`	Возвращает `true`, если `ch` разрешено быть первым символом идентификатора Java, или `false` в противном случае
`static boolean isLetter (char ch)`	Возвращает `true`, если `ch` является буквой, или `false` в противном случае
`static boolean isLetterOrDigit(char ch)`	Возвращает `true`, если `ch` является буквой или цифрой, или `false` в противном случае
`static boolean isLowerCase(char ch)`	Возвращает `true`, если `ch` является буквой в нижнем регистре, или `false` в противном случае
`static boolean isMirrored (char ch)`	Возвращает `true`, если `ch` является зеркальным символом Unicode, т.е. таким, который разворачивается в противоположном направлении для текста, отображаемого справа налево. В противном случае возвращает `false`

Окончание табл. 19.7

Метод	Описание
`static boolean isSpaceChar (char ch)`	Возвращает `true`, если `ch` является символом пробела Unicode, или `false` в противном случае
`static boolean isTitleCase (char ch)`	Возвращает `true`, если `ch` является заглавным символом Unicode, или `false` в противном случае
`static boolean isUnicodeIdentifierPart (char ch)`	Возвращает `true`, если `ch` разрешено быть частью идентификатора Unicode (кроме первого символа), или `false` в противном случае
`static Boolean isUnicodeIdentifierStart (char ch)`	Возвращает `true`, если `ch` разрешено быть первым символом идентификатора Unicode, или `false` в противном случае
`static boolean isUpperCase (char ch)`	Возвращает `true`, если `ch` является буквой в верхнем регистре, или `false` в противном случае
`static boolean isWhitespace(char ch)`	Возвращает `true`, если `ch` является пробельным символом, или `false` в противном случае
`static char toLowerCase (char ch)`	Возвращает эквивалент `ch` в нижнем регистре
`static char toTitleCase (char ch)`	Возвращает заглавный эквивалент `ch`
`static char toUpperCase (char ch)`	Возвращает эквивалент `ch` в верхнем регистре

В классе `Character` определены два метода, `forDigit()` и `digit()`, позволяющие выполнять преобразование между целочисленными значениями и цифрами, которые они представляют:

```
static char forDigit(int num, int radix)
static int digit(char digit, int radix)
```

Метод `forDigit()` возвращает цифровой символ, связанный со значением `num`. Основание системы счисления для преобразования указывается в `radix`. Метод `digit()` возвращает целочисленное значение, которое ассоциировано с заданным символом (который предположительно является цифрой) в соответствии с указанной системой счисления. (Существует вторая форма метода `digit()`, принимающая кодовую точку. Кодовые точки обсуждаются в следующем разделе.)

В классе Character определен еще один метод, compareTo():

```
int compareTo(Character c)
```

Метод compareTo() возвращает нулевое значение, если вызывающий объект и c одинаковые, отрицательное значение, если вызывающий объект меньше c, и положительное значение, если вызывающий объект больше c.

В состав класса Character входит метод по имени getDirectionality(), который можно использовать для выяснения направленности символа, определяемой несколькими константами. В большинстве программ необходимости в применении направленности символов не возникает.

Кроме того, в классе Character переопределены методы equals() и hashCode(), а также предоставлено несколько других методов. Начиная с версии JDK 15, класс Character реализует интерфейс Constable.

Двумя другими классами, связанными с символами, являются Character.Subset, который применяется для описания подмножества Unicode, и Character.UnicodeBlock, содержащий блоки символов Unicode.

Дополнения класса Character для поддержки кодовых точек Unicode

Несколько лет назад в класс Character были внесены важные дополнения, поддерживающие 32-битные символы Unicode. На заре развития Java все символы Unicode могли содержать 16 бит, что соответствует размеру типа char (и размеру значения, инкапсулированного внутри Character), т.к. значения символов находились в диапазоне от 0 до FFFF. Однако набор символов Unicode был расширен и потому требуется более 16 бит. Теперь значения символов могут варьироваться от 0 до 10FFFF.

Рассмотрим три важных термина. *Кодовая точка* — это символ в диапазоне от 0 до 10FFFF. Символы, значения которых превышают FFFF, называются *дополнительными символами. Основная многоязычная плоскость* (basic multilingual plane — BMP) охватывает символы от 0 до FFFF.

Расширение набора символов Unicode стало причиной возникновения фундаментальной проблемы для языка Java. Поскольку дополнительный символ имеет значение больше, чем может содержать char, потребовались средства для обработки дополнительных символов. Проблема была решена двумя способами. Во-первых, для представления дополнительного символа в Java применяются два значения char. Первое значение char называется *старшим суррогатом*, а второе — *младшим суррогатом*. Были предложены методы наподобие codePointAt() для трансляции между кодовыми точками и дополнительными символами.

Во-вторых, несколько ранее существовавших методов в классе Character были перегружены. В перегруженных формах используются данные типа int, а не char. Поскольку тип int достаточен для хранения символа в виде одного значения, его можно применять для хранения любого символа. Например, все методы в табл. 19.7 имеют перегруженные формы, работающие с int, например:

```
static boolean isDigit(int cp)
static boolean isLetter(int cp)
static int toLowerCase(int cp)
```

Помимо методов, перегруженных для приема кодовых точек, в класс Character добавлены методы, обеспечивающие дополнительную поддержку кодовых точек, часть которых описана в табл. 19.8.

Таблица 19.8. Избранные методы, которые обеспечивают поддержку 32-битных кодовых точек Unicode

Метод	Описание
`static int charCount (int cp)`	Возвращает 1, если значение cp может быть представлено как одиночное значение char, и 2, если требуются два значения char
`static int codePointAt (CharSequence chars, int loc)`	Возвращает кодовую точку в позиции, указанной в loc
`static int codePointAt (char[] chars, int loc)`	Возвращает кодовую точку в позиции, указанной в loc
`static int codePointBefore (CharSequence chars, int loc)`	Возвращает кодовую точку в позиции, которая предшествует указанной в loc
`static int codePointBefore (char[] chars, int loc)`	Возвращает кодовую точку в позиции, которая предшествует указанной в loc
`static boolean isBmpCodePoint(int cp)`	Возвращает true, если cp является частью основной многоязычной плоскости, или false в противном случае
`static boolean isHighSurrogate(char ch)`	Возвращает true, если cp содержит допустимый старший суррогатный символ
`static boolean isLowSurrogate(char ch)`	Возвращает true, если cp содержит допустимый младший суррогатный символ
`static boolean isSupplementaryCodePoint (int cp)`	Возвращает true, если cp содержит дополнительный символ
`static boolean isSurrogatePair (char highCh, char lowCh)`	Возвращает true, если highCh и lowCh образуют допустимую суррогатную пару
`static boolean isValidCodePoint(int cp)`	Возвращает true, если cp содержит допустимую кодовую точку
`static char[] toChars (int cp)`	Преобразует кодовую точку cp в эквивалент типа char, который может потребовать двух значений char. Возвращает результат в виде массива

Окончание табл. 19.8

Метод	Описание
`static int toChars (int cp, char[] target, int loc)`	Преобразует кодовую точку `cp` в ее эквивалент типа `char` с сохранением результата в `target`, начиная с позиции `loc`. Возвращает 1, если кодовая точка `cp` может быть представлена одним значением `char`, или 2 в противном случае
`static int toCodePoint (char highCh, char lowCh)`	Преобразует `highCh` и `lowCh` в эквивалентную кодовую точку

Boolean

Класс `Boolean` представляет собой очень тонкую оболочку для булевских значений, которая полезна главным образом, когда необходимо передавать булевскую переменную по ссылке. Он содержит константы `TRUE` и `FALSE`, которые определяют объекты `Boolean` с истинным и ложным значениями. В `Boolean` также определено поле `TYPE`, которое является объектом `Class` для `boolean`. Вот конструкторы класса `Boolean`:

```
Boolean(boolean boolValue)
Boolean(String boolString)
```

В первой версии конструктора параметр `boolValue` должен иметь значение `true` или `false`. Если `boolString` во второй версии конструктора содержит строку `"true"` (в верхнем или нижнем регистре), тогда новый объект `Boolean` получит значение `true`, а в противном случае — `false`. Начиная с JDK 9, эти конструкторы объявлены нерекомендуемыми, а в версии JDK 16 они помечены как устаревшие и подлежащие удалению. Настоятельно рекомендуемой альтернативой является метод `valueOf()`.

В табл. 19.9 кратко описаны часто используемые методы, определенные в классе `Boolean`. Начиная с версии JDK 15, класс `Boolean` также реализует специальный интерфейс `Constable`.

Таблица 19.9. Часто используемые методы класса `Boolean`

Метод	Описание
`boolean booleanValue()`	Возвращает эквивалент `boolean`
`static int compare(boolean b1, boolean b2)`	Возвращает ноль, если `b1` и `b2` содержат одно и то же значение. Возвращает положительное значение, если `b1` равно `true`, а `b2` — `false`, и отрицательное значение в противном случае

Окончание табл. 19.9

Метод	Описание
`int compareTo(Boolean b)`	Возвращает ноль, если вызывающий объект и b содержат одно и то же значение. Возвращает положительное значение, если вызывающий объект равен `true`, а b — `false`, и отрицательное значение в противном случае
`boolean equals (Object boolObj)`	Возвращает `true`, если вызывающий объект эквивалентен `boolObj`, или `false` в противном случае
`static Boolean getBoolean (String propertyName)`	Возвращает `true`, если системное свойство, указанное в `propertyName`, равно `true`, или `false` в противном случае
`int hashCode()`	Возвращает хеш-код для вызывающего объекта
`static int hashCode (boolean boolVal)`	Возвращает хеш-код для `boolVal`
`static boolean logicalAnd(boolean op1, boolean op2)`	Выполняет операцию логического И над `op1` и `op2` и возвращает результат
`static boolean logicalOr (boolean op1, boolean op2)`	Выполняет операцию логического ИЛИ над `op1` и `op2` и возвращает результат
`static boolean logicalXor(boolean op1, boolean op2)`	Выполняет операцию логического исключающего ИЛИ над `op1` и `op2` и возвращает результат
`static boolean parseBoolean(String str)`	Возвращает `true`, если `str` содержит строку `"true"` (регистр символов роли не играет), или `false` в противном случае
`String toString()`	Возвращает строковый эквивалент вызывающего объекта
`static String toString (boolean boolVal)`	Возвращает строковый эквивалент `boolVal`
`static Boolean valueOf (boolean boolVal)`	Возвращает булевский эквивалент `boolVal`
`static Boolean valueOf (String boolString)`	Возвращает `true`, если `boolString` содержит строку `"true"` (регистр символов роли не играет), или `false` в противном случае

Void

Класс Void имеет одно поле TYPE, которое содержит ссылку на объект Class для типа void. Создавать экземпляры этого класса не придется.

Process

Абстрактный класс Process инкапсулирует *процесс*, т.е. исполняемую программу. Он используется главным образом как суперкласс для типов объектов, созданных методом exec() в классе Runtime или методом start() в классе ProcessBuilder. Избранные методы Process кратко описаны в табл. 19.10. Начиная с версии JDK 9, дескриптор процесса можно получить в виде экземпляра ProcessHandle вместе с информацией о процессе, инкапсулированной в экземпляре ProcessHandle.Info, что обеспечивает дополнительный уровень контроля. Особо интересной частью данной информации является количество процессорного времени, получаемое процессом, для получения которого необходимо вызвать метод totalCpuDuration(), определенный в ProcessHandle.Info. Другие крайне полезные сведения можно получить, вызвав метод isAlive() экземпляра ProcessHandle, который возвратит true, если процесс все еще выполняется. Начиная с версии JDK 17, класс Process также предоставляет методы inputReader(), errorReader() и outputWriter().

Таблица 19.10. Часто используемые методы класса Process

Метод	Описание
`Stream<ProcessHandle> children()`	Возвращает поток, который содержит объекты ProcessHandle, представляющие непосредственные дочерние процессы вызывающего процесса
`Stream<ProcessHandle> descendants()`	Возвращает поток, который содержит объекты ProcessHandle, представляющие непосредственные дочерние процессы вызывающего процесса, плюс все его потомки
`void destroy()`	Прекращает работу процесса
`Process destroyForcibly()`	Принудительно прекращает работу вызывающего процесса. Возвращает ссылку на процесс
`int exitValue()`	Возвращает код завершения, полученный из подчиненного процесса
`InputStream getErrorStream()`	Возвращает поток ввода, который читает данные из потока вывода err процесса

Окончание табл. 19.10

Метод	Описание
`InputStream getInputStream()`	Возвращает поток ввода, который читает данные из потока вывода `out` процесса
`OutputStream getOutputStream()`	Возвращает поток вывода, который записывает данные в поток ввода `in` процесса
`ProcessHandle.Info info()`	Возвращает информацию о процессе в виде объекта `ProcessHandle.Info`
`boolean isAlive()`	Возвращает `true`, если вызывающий процесс все еще активен, или `false` в противном случае
`CompletableFuture <Process> onExit()`	Возвращает объект `CompletableFuture` для вызывающего процесса, который может использоваться для выполнения задач при прекращении работы
`long pid()`	Возвращает идентификатор процесса, ассоциированный с вызывающим процессом
`boolean supportsNormalTermination()`	Определяет, к какому прекращению работы приведет вызов метода `destroy()` — нормальному или принудительному. Возвращает `true`, если прекращение работы будет нормальным, или `false` в противном случае
`ProcessHandle toHandle()`	Возвращает дескриптор вызывающего процесса в виде объекта `ProcessHandle`
`int waitFor() throws InterruptedException`	Возвращает код завершения, возвращенный процессом. Метод не возвращает управление до тех пор, пока не прекратит работу процесс, для которого он был вызван
`boolean waitFor (long waitTime, TimeUnit timeUnit) throws InterruptedException`	Ожидает окончания вызывающего процесса. Время ожидания задается параметром `waitTime` в единицах, указанных с помощью `timeUnit`. Возвращает `true`, если процесс закончился, и `false`, если истекло время ожидания

Runtime

Класс `Runtime` инкапсулирует исполняющую среду. Создать экземпляр объекта `Runtime` невозможно, но можно получить ссылку на текущий объект исполняющей среды, вызвав статический метод `Runtime.getRuntime()`.

После получения ссылки на текущий объект исполняющей среды можно вызывать несколько методов, управляющих состоянием и поведением виртуальной машины Java. В не заслуживающем доверия коде обычно нельзя вызывать какие-либо методы Runtime без генерации исключения SecurityException. Избранные методы класса Runtime кратко описаны в табл. 19.11.

Таблица 19.11. Часто используемые методы класса Runtime

Метод	Описание
`void addShutdownHook (Thread thrd)`	Регистрирует thrd в качестве потока, который будет запущен после прекращения работы виртуальной машины Java
`Process exec (String progName) throws IOException`	Выполняет программу, указанную в progName, как отдельный процесс. Возвращает объект типа Process, который описывает новый процесс
`Process exec (String progName, String[] environment) throws IOException`	Выполняет программу, указанную в progName, как отдельный процесс в среде, заданной с помощью environment. Возвращает объект типа Process, который описывает новый процесс
`Process exec(String[] comLineArray) throws IOException`	Выполняет командную строку, указанную в comLineArray, как отдельный процесс. Возвращает объект типа Process, который описывает новый процесс
`Process exec(String[] comLineArray, String[] environment) throws IOException`	Выполняет командную строку, указанную в comLineArray, как отдельный процесс в среде, заданной с помощью environment. Возвращает объект типа Process, который описывает новый процесс
`void exit(int exitCode)`	Останавливает выполнение и возвращает значение exitCode родительскому процессу. По соглашению 0 означает нормальное прекращение работы, а все остальные значения указывают на некоторую форму ошибки
`long freeMemory()`	Возвращает приблизительное количество байтов свободной памяти, доступной исполняющей среде Java
`void gc()`	Инициирует сборку мусора
`static Runtime getRuntime()`	Возвращает текущий объект Runtime

Окончание табл. 19.11

Метод	Описание
`void halt(int code)`	Немедленно прекращает работу виртуальной машины Java. Потоки завершения и финализации не запускаются. Вызывающему процессу возвращается значение code
`void load(String libraryFileName)`	Загружает динамическую библиотеку, файл которой указан в `libraryFileName` и должен включать полный путь
`void loadLibrary(String libraryName)`	Загружает динамическую библиотеку, имя которой ассоциировано с `libraryName`
`Boolean removeShutdownHook (Thread thrd)`	Удаляет `thrd` из списка потоков, запускаемых при прекращении работы виртуальной машины Java. Возвращает `true` в случае успеха, т.е. если поток был удален
`void runFinalization()`	Инициирует вызовы методов `finalize()` неиспользуемых, но еще не уничтоженных объектов
`long totalMemory()`	Возвращает общее количество байтов памяти, доступной программе
`static Runtime.Version version()`	Возвращает используемую версию Java. Детали ищите в разделе, посвященном `Runtime.Version` далее в главе

Давайте рассмотрим одно из наиболее интересных применений класса Runtime: выполнение дополнительных процессов.

Выполнение других программ

В безопасных средах средства Java можно использовать для выполнения других тяжеловесных процессов (т.е. программ) под управлением многозадачной операционной системы (ОС). Несколько форм метода `exec()` позволяют указать программу, подлежащую запуску, и ее входные параметры. Метод `exec()` возвращает объект `Process`, с помощью которого осуществляется последующее управление тем, как текущая программа на Java взаимодействует с новым запущенным процессом. Поскольку Java может работать на разнообразных платформах и под управлением различных ОС, метод `exec()` по существу зависит от среды.

В приведенном ниже примере метод `exec()` применяется для запуска простого текстового редактора Windows по имени notepad. Очевидно, что код примера должен запускаться в среде Windows.

```
// Демонстрация работы exec().
class ExecDemo {
  public static void main(String[] args) {
    Runtime r = Runtime.getRuntime();
    Process p = null;

    try {
      p = r.exec("notepad");
    } catch (Exception e) {
      System.out.println("Ошибка при выполнении notepad.");
    }
  }
}
```

Существует ряд альтернативных форм `exec()`, но показанной в примере формы часто оказывается вполне достаточно. Объектом `Process`, который возвращает метод `exec()`, можно управлять с помощью методов `Process` после того, как новая программа начнет выполняться. Метод `destroy()` позволяет уничтожить подпроцесс. Метод `waitFor()` заставляет программу ожидать завершения подпроцесса. Метод `exitValue()` возвращает значение, возвращенное подпроцессом после его завершения. Обычно это 0, если никаких проблем не возникало. Вот предыдущий пример использования `exec()`, модифицированный для ожидания завершения работающего процесса:

```
// Ожидать окончания работы notepad.
class ExecDemoFini {
  public static void main(String[] args) {
    Runtime r = Runtime.getRuntime();
    Process p = null;

    try {
      p = r.exec("notepad");
      p.waitFor();
    } catch (Exception e) {
      System.out.println("Ошибка при выполнении notepad.");
    }
    System.out.println("Программа notepad возвратила " + p.exitValue());
  }
}
```

Во время функционирования подпроцесса можно выполнять запись и чтение из его стандартных потоков ввода и вывода. Методы `getOutputStream()` и `getInputStream()` возвращают дескрипторы стандартных потоков `in` и `out` подпроцесса. Начиная с версии JDK 17, в качестве альтернативы можно применять методы `outputWriter()` и `inputReader()` для получения средств записи и чтения. (Ввод-вывод подробно обсуждается в главе 22.)

Runtime.Version

Внутри `Runtime.Version` инкапсулируется информация о версии (включая номер версии), относящаяся к среде Java. Вызвав `Runtime.version()`, можно получить экземпляр `Runtime.Version` для текущей платформы.

Первоначально добавленный в JDK 9 класс `Runtime.Version` в версии JDK 10 был существенно изменен, чтобы лучше соответствовать более быстрой, основанной на времени частоте выпусков. Как обсуждалось ранее в книге, начиная с JDK 10, ожидается, что выпуск функциональных средств будет происходить по строгому графику, а интервал между выпусками составит полгода. `Runtime.Version` — класс, основанный на значениях. (Описание классов, основанных на значениях, приводилось в главе 13.)

В прошлом для номера версии JDK использовался хорошо известный подход `старший.младший`, но такой механизм не соответствовал графику выпуска, основанному на времени. В результате элементам номера версии придавалось другое значение. На сегодняшний день в первых четырех элементах указываются *счетчики*, расположенные в следующем порядке: счетчик выпуска функциональных средств, счетчик промежуточного выпуска, счетчик выпуска обновлений и счетчик выпуска исправлений. Числа отделяются друг от друга точками. Тем не менее, завершающиеся нули вместе с предшествующими им точками удаляются. Хотя могут быть включены и дополнительные элементы, предопределенный смысл имеется только у первых четырех.

Счетчик выпусков функциональных средств задает номер выпуска и обновляется с каждым выпуском функциональных средств. Чтобы сгладить переход от схемы предыдущей версии, счетчик выпусков функциональных средств начинался с 10. Таким образом, счетчик выпусков функциональных средств для JDK 10 равен 10, для JDK 11 — 11 и т.д.

Счетчик промежуточных выпусков указывает номер выпуска, который происходит между выпусками функциональных средств. На момент написания главы значение счетчика промежуточных выпусков будет равно нулю, т.к. ожидается, что промежуточные выпуски не будут частью увеличенной частоты выпусков. (Он определен для возможного применения в будущем.) Промежуточный выпуск не будет приводить к критическим изменениям в наборе функциональных средств JDK. Счетчик выпусков обновлений задает номер выпуска, который решает проблемы безопасности и возможно другие проблемы. Счетчик выпусков исправлений указывает номер выпуска, устраняющего серьезную ошибку, которую необходимо исправить как можно скорее. С каждым выходом нового выпуска функциональных средств счетчики промежуточных выпусков, обновлений и исправлений обнуляются.

Полезно отметить, что описанный выше номер версии является необходимым компонентом *строки версии*, но в строку также могут быть включены необязательные элементы. Например, строка версии может содержать информацию о предварительной версии. Необязательные элементы следуют за номером версии в строке версии.

В версии JDK 10 класс `Runtime.Version` был обновлен и теперь включает следующие методы, которые поддерживают новые значения счетчиков выпусков функциональных средств, промежуточных выпусков, обновлений и исправлений:

```
int feature()
int interim()
int update()
int patch()
```

Каждый метод возвращает целочисленное значение, которое представляет соответствующий счетчик. Ниже приведена короткая программа, демонстрирующая их использование:

```
// Демонстрация использования счетчиков выпусков Runtime.Version.
class VerDemo {
  public static void main(String[] args) {
    Runtime.Version ver = Runtime.version();

    // Отобразить индивидуальные счетчики.
    System.out.println("Счетчик выпусков функциональных средств: "
        + ver.feature());
    System.out.println("Счетчик промежуточных выпусков: "
        + ver.interim());
    System.out.println("Счетчик выпусков обновлений: " + ver.update());
    System.out.println("Счетчик выпусков исправлений: " + ver.patch());
  }
}
```

В результате перехода на выпуски, основанные на времени, в `Runtime.Version` объявлены устаревшими методы `major()`, `minor()` и `security()`. Ранее они возвращали основной номер версии, дополнительный номер версии и номер обновления безопасности. Указанные значения были заменены номерами выпусков функциональных средств, промежуточных выпусков и выпусков обновлений.

Кроме только что рассмотренных методов в `Runtime.Version` есть методы, предназначенные для получения различных фрагментов необязательных данных. Например, вызвав метод `build()`, можно получить номер сборки при его наличии. Предварительная информация, если она есть, возвращается методом `pre()`. Также может присутствовать другая необязательная информация, которую можно получить с помощью метода `optional()`. Для сравнения версий применяются методы `compareTo()` и `compareToIgnoreOptional()`. Для определения равенства версий можно использовать методы `equals()` и `equalsIgnoreOptional()`. Метод `version()` возвращает список номеров версий. Вызвав метод `parse()`, можно преобразовать допустимую строку версии в объект `Runtime.Version`.

ProcessBuilder

Класс `ProcessBuilder` предоставляет еще один способ запуска процессов (т.е. программ) и управления ими. Как объяснялось ранее, все процессы представлены классом `Process`, и процесс может быть запущен с помощью `Runtime.exec()`. Класс `ProcessBuilder` предлагает более высокую степень контроля над процессами. Например, он разрешает установку текущего рабочего каталога.

В классе ProcessBuilder определены следующие конструкторы:

```
ProcessBuilder(List<String> args)
ProccessBuilder(String ... args)
```

Здесь args — список аргументов, которые задают имя подлежащей запуску программы вместе со всеми необходимыми аргументами командной строки. В первом конструкторе аргументы передаются в списке List. Во втором они указываются через аргументы переменной длины. Методы, определенные в ProcessBuilder, кратко описаны в табл. 19.12.

В табл. 19.12 обратите внимание на методы, работающие с классом ProcessBuilder.Redirect. Этот абстрактный класс инкапсулирует источник или цель ввода-вывода, связанную с подпроцессом. Помимо прочего такие методы позволяют перенаправить источник или цель операций ввода-вывода.

Таблица 19.12. Часто используемые методы класса ProcessBuilder

Метод	Описание
`List<String> command()`	Возвращает ссылку на объект List, который содержит имя программы и ее аргументы. Изменения в этом списке влияют на вызывающий объект
`ProcessBuilder command (List<String> args)`	Устанавливает имя программы и ее аргументы так, как указанно в args. Изменения в этом списке влияют на вызывающий объект. Возвращает ссылку на вызывающий объект
`ProcessBuilder command (String ... args)`	Устанавливает имя программы и ее аргументы так, как указанно в args. Возвращает ссылку на вызывающий объект
`File directory()`	Возвращает текущий рабочий каталог вызывающего объекта. Возвращаемое значение будет равно null, если каталог совпадает с каталогом программы на Java, которая запустила процесс
`ProcessBuilder directory (File dir)`	Устанавливает текущий рабочий каталог вызывающего объекта. Возвращает ссылку на вызывающий объект
`Map<String, String> environment()`	Возвращает переменные среды, ассоциированные с вызывающим объектом, в виде пар "ключ-значение"

Продолжение табл. 19.12

Метод	Описание
`ProcessBuilder inheritIO()`	Заставляет вызванный процесс использовать тот же источник и цель для стандартных потоков ввода-вывода, что и вызывающий процесс
`ProcessBuilder.Redirect redirectError()`	Возвращает цель для стандартного потока ошибок как объект `ProcessBuilder.Redirect`
`ProcessBuilder redirectError(File f)`	Устанавливает цель для стандартного потока ошибок в указанный файл. Возвращает ссылку на вызывающий объект
`ProcessBuilder redirectError (ProcessBuilder.Redirect target)`	Устанавливает цель для стандартного потока ошибок как указано в `target`. Возвращает ссылку на вызывающий объект
`boolean redirectErrorStream()`	Возвращает `true`, если стандартный поток ошибок был перенаправлен в стандартный поток вывода. Возвращает `false`, если потоки являются отдельными
`ProcessBuilder redirectErrorStream (boolean merge)`	Если значение `merge` равно `true`, тогда стандартный поток ошибок перенаправляется в стандартный поток вывода. Если значение `merge` равно `false`, то потоки являются отдельными — состояние, принятое по умолчанию. Возвращает ссылку на вызывающий объект
`ProcessBuilder.Redirect redirectInput()`	Возвращает источник для стандартного потока ввода в виде объекта `ProcessBuilder.Redirect`
`ProcessBuilder redirectInput(File f)`	Устанавливает источник для стандартного потока ввода в указанный файл. Возвращает ссылку на вызывающий объект
`ProcessBuilder redirectInput (ProcessBuilder.Redirect source)`	Устанавливает источник для стандартного потока ввода как указано в `source`. Возвращает ссылку на вызывающий объект
`ProcessBuilder.Redirect redirectOutput()`	Возвращает цель для стандартного потока вывода в виде объекта `ProcessBuilder.Redirect`
`ProcessBuilder redirectOutput(File f)`	Устанавливает цель для стандартного потока вывода в указанный файл. Возвращает ссылку на вызывающий объект

Окончание табл. 19.12

Метод	Описание
`ProcessBuilder redirectOutput (ProcessBuilder.Redirect target)`	Устанавливает цель для стандартного потока вывода как указано в target. Возвращает ссылку на вызывающий объект
`Process start() throws IOException`	Начинает процесс, указанный вызывающим объектом. Другими словами, запускает указанную программу
`static List<Process> startPipeline(List <ProcessBuilder> pbList) throws IOException`	Направляет процессы по конвейеру в pbList

Например, можно реализовать перенаправление в файл, вызвав `to()`, перенаправление из файла с помощью `from()` и добавление в файл посредством `appendTo()`. Объект `File`, связанный с файлом, можно получить, вызвав метод `file()`. Упомянутые методы перечислены ниже:

```
static ProcessBuilder.Redirect to(File f)
static ProcessBuilder.Redirect from(File f)
static ProcessBuilder.Redirect appendTo(File f)
File file()
```

В классе `ProcessBuilder.Redirect` поддерживается еще один метод — `type()`, который возвращает значение перечислимого типа `ProcessBuilder.Redirect.Type`, описывающего вид перенаправления. В нем определены следующие значения: `APPEND`, `INHERIT`, `PIPE`, `READ` и `WRITE`. Кроме того, в `ProcessBuilder.Redirect` определены константы `INHERIT`, `PIPE` и `DISCARD`.

Для создания процесса с применением `ProcessBuilder` нужно просто создать экземпляр `ProcessBuilder`, указав имя программы и все необходимые аргументы. Чтобы начать выполнение программы, понадобится вызвать метод `start()` для нового экземпляра `ProcessBuilder`. Вот пример, в котором запускается текстовый редактор notepad. Обратите внимание, что в качестве аргумента указывается имя файла, подлежащего редактированию.

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

    try {
      ProcessBuilder proc =
        new ProcessBuilder("notepad.exe", "testfile");
      proc.start();
    } catch (Exception e) {
      System.out.println("Ошибка при выполнении notepad.");
    }
  }
}
```

System

Класс System содержит набор статических методов и переменных. Стандартные потоки ввода, вывода и ошибок исполняющей среды Java хранятся в переменных in, out и err. В табл. 19.13 кратко описаны методы класса System, которые не являются нерекомендуемыми. Многие методы генерируют исключение SecurityException, если операция не разрешена диспетчером безопасности. Однако имейте в виду, что в JDK 17 диспетчер безопасности помечен как устаревший и подлежащий удалению.

Таблица 19.13. Методы класса System, которые не объявлены нерекомендуемыми

Метод	Описание
`static void arraycopy (Object source, int sourceStart, Object target, int targetStart, int size)`	Копирует массив. Подлежащий копированию массив передается в source, а индекс, с которого начинается копирование в source, указывается в sourceStart. Массив, получающий копию, передается в target, а индекс, с которого копия начнется в target, указывается в targetStart. Количество копируемых элементов задается в size
`static String clearProperty (String which)`	Удаляет переменную среды, указанную в which. Возвращает предыдущее значение, ассоциированное с which
`static Console console()`	Возвращает консоль, связанную с виртуальной машиной Java. Если машина JVM в текущий момент не имеет консоли, тогда возвращается null
`static long currentTimeMillis()`	Возвращает текущее время в миллисекундах, прошедших с полуночи 1 января 1970 года
`static void exit(int exitCode)`	Останавливает выполнение и возвращает значение exitCode родительскому процессу (обычно ОС). По соглашению 0 означает нормальное завершение, а все остальные значения указывают на некоторую форму ошибки
`static void gc()`	Инициирует сборку мусора
`static Map<String, String> getenv()`	Возвращает объект Map, который содержит текущие переменные среды и их значения
`static String getenv(String which)`	Возвращает значение, ассоциированное с переменной среды, которая передается в which
`static System.Logger getLogger (String logName)`	Возвращает ссылку на объект, который можно использовать для ведения журнала программы. Имя средства ведения журнала передается в logName

Продолжение табл. 19.13

Метод	Описание
`static System.Logger getLogger (String logName, ResourceBundle rb)`	Возвращает ссылку на объект, который можно использовать для ведения журнала программы. Имя средства ведения журнала передается в `logName`. Локализация поддерживается пакетом ресурсов, переданным в `rb`
`static Properties getProperties()`	Возвращает свойства, ассоциированные с исполняющей средой Java. (Класс `Properties` обсуждается в главе 20.)
`static String getProperty (String which)`	Возвращает свойство, ассоциированное с `which`. Если желаемое свойство не найдено, тогда возвращается `null`
`static String getProperty (String which, String default)`	Возвращает свойство, ассоциированное с `which`. Если желаемое свойство не найдено, тогда возвращается `default`
`static int identityHashCode (Object obj)`	Возвращает хеш-код идентичности для `obj`
`static Channel inheritedChannel() throws IOException`	Возвращает канал, унаследованный виртуальной машиной Java, или `null`, если унаследованных каналов нет
`static String lineSeparator()`	Возвращает строку, которая содержит символы разделителя строчек
`static void load(String libraryFileName)`	Загружает динамическую библиотеку, файл которой указан в `libraryFileName` и должен содержать полный путь
`static void loadLibrary(String libraryName)`	Загружает динамическую библиотеку, имя которой ассоциировано с `libraryName`
`static String mapLibraryName (String lib)`	Возвращает специфичное для платформы имя библиотеки, указанной в `lib`
`static long nanoTime()`	Получает самый точный таймер в системе и возвращает его значение в наносекундах с некоторой произвольной начальной точки. Точность таймера нераспознаваема
`static void runFinalization()`	Инициирует вызовы методов `finalize()` неиспользуемых, но еще не уничтоженных объектов

Окончание табл. 19.13

Метод	Описание
`static void setErr(PrintStream eStream)`	Устанавливает стандартный поток err в eStream
`static void setIn(InputStream iStream)`	Устанавливает стандартный поток in в iStream
`static void setOut(PrintStream oStream)`	Устанавливает стандартный поток out в oStream
`static void setProperties (Properties sysProperties)`	Устанавливает текущие свойства системы как указано в sysProperties
`static String setProperty(String which, String v)`	Присваивает значение v свойству, имя которого указано в which

Давайте взглянем на распространенные применения класса System.

Использование currentTimeMillis() для хронометража выполнения программы

Одно из особенно интересных применений класса System связано с использованием метода currentTimeMillis() для определения, сколько времени занимает выполнение различных частей программы. Метод currentTimeMillis() возвращает текущее время в миллисекундах, прошедших с полуночи 1 января 1970 года. Для хронометража раздела программы сохраните такое значение прямо перед началом интересующего раздела. Немедленно после завершения раздела снова вызовите currentTimeMillis(). Затраченное время вычисляется как разность времени окончания и времени начала, что демонстрируется ниже в программе:

```
// Хронометраж выполнения программы.
class Elapsed {
  public static void main(String[] args) {
    long start, end;

    System.out.println("Хронометраж цикла for от 0 до 100 000 000");

    // Хронометрировать цикл for от 0 до 100 000 000.
    start = System.currentTimeMillis();  // получить время начала
    for(long i=0; i < 100000000L; i++);
    end = System.currentTimeMillis();    // получить время окончания

    System.out.println("Затраченное время: " + (end-start));
  }
}
```

Вот результаты запуска (не забывайте, что полученные вами результаты могут отличаться):

```
Хронометраж цикла for от 0 до 100 000 000
Затраченное время: 10
```

Если в вашей системе имеется таймер с наносекундной точностью, тогда можете переписать предыдущую программу так, чтобы в ней применялся метод `nanoTime()`, а не `currentTimeMillis()`. Например, ниже показана ключевая часть программы, переписанная с использованием `nanoTime()`:

```
start = System.nanoTime();        // получить время начала
for(long i=0; i < 100000000L; i++);
end = System.nanoTime();          // получить время окончания
```

Использование `arraycopy()`

Метод `arraycopy()` можно применять для быстрого копирования массива любого типа из одного места в другое, что будет намного быстрее, чем эквивалентный цикл, реализованный на Java. Далее приведен пример копирования двух массивов с помощью `arraycopy()`. Сначала `a` копируется в `b`, после чего в массиве `a` выполняется сдвиг на один элемент вправо, а в массиве `b` — сдвиг на один элемент влево.

```
// Использование arraycopy().

class ACDemo {
  static byte[] a = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74 };
  static byte[] b = { 77, 77, 77, 77, 77, 77, 77, 77, 77, 77 };

  public static void main(String[] args) {
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
    System.arraycopy(a, 0, b, 0, a.length);
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
    System.arraycopy(a, 0, a, 1, a.length - 1);
    System.arraycopy(b, 1, b, 0, b.length - 1);
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
  }
}
```

В следующем выводе видно, что копировать можно между одним и тем же источником и получателем в обоих направлениях:

```
a = ABCDEFGHIJ
b = MMMMMMMMMM
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCDEFGHI
b = BCDEFGHIJJ
```

Свойства среды

На момент написания главы были доступны перечисленные ниже свойства:

file.separator	java.vendor	java.vm.version
java.class.path	java.vendor.url	line.separator
java.class.version	java.vendor.version	native.encoding
java.compiler	java.version	os.arch
java.home	java.version.date	os.name
java.io.tmpdir	java.vm.name	os.version
java.library.path	java.vm.specification.name	path.separator
java.specification.name	java.vm.specification.vendor	user.dir
java.specification.vendor	java.vm.specification.version	user.home
java.specification.version	java.vm.vendor	user.name

Вызвав метод System.getProperty(), можно получить значения различных переменных среды. Например, в следующем коде отображается путь к текущему пользовательскому каталогу:

```
class ShowUserDir {
  public static void main(String[] args) {
    System.out.println(System.getProperty("user.dir"));
  }
}
```

System.Logger и System.LoggerFinder

Интерфейс System.Logger и класс System.LoggerFinder поддерживают журнал программы. Средство ведения журнала можно найти с использованием System.getLogger(), а System.Logger предоставляет интерфейс для этого средства.

Object

В части I упоминалось о том, что Object является суперклассом для всех остальных классов. В Object определены методы, кратко описанные в табл. 19.14, которые доступны каждому объекту.

Таблица 19.14. Методы, определенные в классе Object

Метод	Описание
Object clone() throws CloneNotSupportedException	Создает новый объект, который будет таким же, как вызывающий объект
boolean equals(Object object)	Возвращает true, если вызывающий объект эквивалентен объекту object

Окончание табл. 19.14

Метод	Описание
`void finalize() throws Throwable`	Стандартный метод `finalize()`, который вызывается перед уничтожением неиспользуемого объекта. (В версии JDK 9 он объявлен нерекомендуемым.)
`final Class<?> getClass()`	Получает объект `Class`, который описывает вызывающий объект
`int hashCode()`	Возвращает хеш-код, ассоциированный с вызывающим объектом
`final void notify()`	Возобновляет выполнение потока, который ожидает на вызывающем объекте
`final void notifyAll()`	Возобновляет выполнение всех потоков, которые ожидают на вызывающем объекте
`String toString()`	Возвращает строку, описывающую объект
`final void wait() throws InterruptedException`	Организует ожидание на другом потоке выполнения
`final void wait (long milliseconds) throws InterruptedException`	Организует ожидание на протяжении указанного в `milliseconds` времени в миллисекундах на другом потоке выполнения
`final void wait (long milliseconds, int nanoseconds) throws InterruptedException`	Организует ожидание на протяжении указанного в `milliseconds` и `nanoseconds` времени в миллисекундах и наносекундах на другом потоке выполнения

Использование метода `clone()` и интерфейса `Cloneable`

Большинство методов, определенных в классе `Object`, обсуждаются в других местах книги, но один из них заслуживает особого внимания: `clone()`. Метод `clone()` создает дубликат объекта, для которого он вызывается. Клонировать можно только классы, реализующие интерфейс `Cloneable`.

В интерфейсе `Cloneable` члены не определены. Он применяется для указания о том, что класс позволяет сделать точную копию объекта (т.е. *клон*). Если вы по-

пытаетесь вызвать метод clone() для класса, который не реализует Cloneable, тогда сгенерируется исключение CloneNotSupportedException. При создании клона конструктор клонируемого объекта не вызывается. Согласно реализации внутри Object клон — это просто точная копия оригинала.

Клонирование — потенциально опасное действие, поскольку оно может привести к непредвиденным побочным эффектам. Например, если клонируемый объект содержит ссылочную переменную по имени obRef, то при создании клона obRef в клоне будет ссылаться на тот же объект, что и obRef в оригинале. Если клон изменяет содержимое объекта, на который ссылается obRef, то оно также изменится для исходного объекта. А вот еще один пример: если объект открывает поток ввода-вывода, после чего клонируется, то два объекта будут способны работать с одним и тем же потоком. Более того, если один из этих объектов закроет поток, то другой объект все еще может попытаться выполнить запись в него, что приведет к ошибке. В некоторых случаях для решения проблем такого рода потребуется переопределить метод clone(), определенный в Object.

Из-за того, что клонирование может приводить к проблемам, метод clone() объявлен как защищенный внутри Object. Отсюда следует, что метод clone() потребуется либо вызывать из метода, который определен в классе, реализующем интерфейс Cloneable, либо явно переопределять его в этом классе, делая его открытым. Давайте рассмотрим пример применения каждого подхода.

В следующей программе класс TestClone реализует интерфейс Cloneable и определяет метод cloneTest(), который вызывает clone() из Object:

```
// Демонстрация использования метода clone().
class TestClone implements Cloneable {
  int a;
  double b;
  // Этот метод вызывает clone() из Object.
  TestClone cloneTest() {
    try {
      // Вызвать clone() из Object.
      return (TestClone) super.clone();
    } catch(CloneNotSupportedException e) {
      System.out.println("Клонирование не разрешено.");
      return this;
    }
  }
}
class CloneDemo {
  public static void main(String[] args) {
    TestClone x1 = new TestClone();
    TestClone x2;
    x1.a = 10;
    x1.b = 20.98;
    x2 = x1.cloneTest();  // клонировать x1
    System.out.println("x1: " + x1.a + " " + x1.b);
    System.out.println("x2: " + x2.a + " " + x2.b);
  }
}
```

Здесь метод cloneTest() вызывает clone() из Object и возвращает результат. Обратите внимание, что объект, возвращаемый методом clone(), должен быть приведен к соответствующему типу (TestClone).

В показанном далее примере метод clone() переопределяется, чтобы его можно было вызывать в коде за пределами класса, где он определен. Для этого его спецификатором доступа должен быть public:

```
// Переопределение метода clone().
class TestClone implements Cloneable {
  int a;
  double b;

  // Метод clone() теперь переопределен и является открытым.
  public Object clone() {
    try {
      // Вызвать clone() из Object.
      return super.clone();
    } catch(CloneNotSupportedException e) {
      System.out.println("Клонирование не разрешено.");
      return this;
    }
  }
}

class CloneDemo2 {
  public static void main(String[] args) {
    TestClone x1 = new TestClone();
    TestClone x2;

    x1.a = 10;
    x1.b = 20.98;

    // Здесь метод clone() вызывается напрямую.
    x2 = (TestClone) x1.clone();

    System.out.println("x1: " + x1.a + " " + x1.b);
    System.out.println("x2: " + x2.a + " " + x2.b);
  }
}
```

Бывает так, что побочные эффекты, вызванные клонированием, поначалу заметить нелегко. Вполне может казаться, что класс безопасен для клонирования, хотя фактически это не так. В общем случае вы не должны обеспечивать реализацию интерфейса Cloneable любым классом без веской причины.

Class

Класс Class инкапсулирует состояние времени выполнения класса или интерфейса. Объекты типа Class создаются автоматически при загрузке классов. Явно объявлять объект Class невозможно. Как правило, объект Class получается путем вызова метода getClass(), определенного в Object. Класс Class является обобщенным и объявлен следующим образом:

```
class Class<T>
```

В T указывается тип представляемого класса или интерфейса. В табл. 19.15 кратко описаны избранные методы, определенные в Class. Обратите внимание на метод getModule(), который относится к поддержке средства модулей, появившейся в JDK 9. Класс Class реализует несколько интерфейсов, в том числе Constable и TypeDescriptor.

Таблица 19.15. Часто используемые методы класса Class

Метод	Описание
`static Class<?> forName(Module mod, String name)`	Возвращает объект Class, соответствующий его полному имени и модулю, в котором он находится
`static Class<?> forName(String name) throws ClassNotFoundException`	Возвращает объект Class по его полному имени
`static Class<?> forName(String name, boolean how, ClassLoader ldr) throws ClassNotFoundException`	Возвращает объект Class по его полному имени. Объект загружается с использованием загрузчика, указанного в ldr. Если how равно true, то объект инициализируется, а если false, то нет
`<A extends Annotation> A getAnnotation (Class<A> annoType)`	Возвращает объект Annotation, содержащий аннотацию, которая ассоциирована с вызывающим объектом и имеет тип annoType
`Annotation[] getAnnotations()`	Получает все аннотации, ассоциированные с вызывающим объектом, и сохраняет их в массиве объектов типа Annotation. Возвращает ссылку на результирующий массив
`<A extends Annotation> A[] getAnnotationsByType( Class<A> annoType)`	Возвращает массив аннотаций (включающий повторяющиеся аннотации) типа annoType, которые ассоциированы с вызывающим объектом
`Class<?>[] getClasses()`	Возвращает объект Class для каждого открытого класса и интерфейса, который является членом класса, представленного вызывающим объектом
`ClassLoader getClassLoader()`	Возвращает объект ClassLoader, который загрузил класс или интерфейс

Продолжение табл. 19.15

Метод	Описание
`Constructor<T> getConstructor(Class<?> ... paramTypes) throws NoSuchMethodException, SecurityException`	Возвращает объект Constructor, представляющий конструктор для класса, к которому относится вызывающий объект, с типами параметров, указанными в paramTypes
`Constructor<?>[] getConstructors() throws SecurityException`	Получает объекты Constructor для всех открытых конструкторов класса, представленного вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на результирующий массив
`Annotation[] getDeclaredAnnotations()`	Получает объект Annotation для всех аннотаций, объявленных вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на результирующий массив. (Унаследованные аннотации игнорируются.)
`<A extends Annotation> A[] getDeclaredAnnotationsByType( Class<A> annoType)`	Возвращает массив неунаследованных аннотаций (содержащий также повторяющиеся аннотации) типа annoType, которые ассоциированы с вызывающим объектом
`Constructor<?>[] getDeclaredConstructors() throws SecurityException`	Получает объекты Constructor для всех конструкторов, которые объявлены в классе, представленном вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на результирующий массив. (Конструкторы суперклассов игнорируются.)
`Field[] getDeclaredFields() throws SecurityException`	Получает объекты Field для всех полей, которые объявлены в классе или интерфейсе, представленном вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на результирующий массив. (Унаследованные поля игнорируются.)

Продолжение табл. 19.15

Метод	Описание
`Method[] getDeclaredMethods() throws SecurityException`	Получает объекты Method для всех методов, которые объявлены в классе или интерфейсе, представленном вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на результирующий массив. (Унаследованные методы игнорируются.)
`Field getField (String fieldName) throws NoSuchMethodException, SecurityException`	Возвращает объект Field с открытым полем, которое указано в fieldName, для класса или интерфейса, представленного вызывающим объектом
`Field[] getFields() throws SecurityException`	Получает объекты Field для всех открытых полей, которые объявлены в классе или интерфейсе, представленном вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на результирующий массив
`Class<?>[] getInterfaces()`	При вызове на объекте, представляющем класс, метод возвращает массив интерфейсов, реализованных данным классом. При вызове на объекте, представляющем интерфейс, метод возвращает массив интерфейсов, расширяющих данный интерфейс
`Method getMethod (String methName, Class<?> ... paramTypes) throws NoSuchMethodException, SecurityException`	Возвращает объект Method, представляющий открытый метод, который указан в methName и имеет типы параметров, заданные в paramTypes, в классе или интерфейсе, представленном вызывающим объектом
`Method[] getMethods() throws SecurityException`	Получает объекты Method для всех открытых методов, которые объявлены в классе или интерфейсе, представленном вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на результирующий массив
`Module getModule()`	Возвращает объект Module, представляющий модуль, в котором находится тип класса вызывающего объекта

Окончание табл. 19.15

Метод	Описание
`String getName()`	Возвращает полное имя класса или интерфейса типа, представленного вызывающим объектом
`String getPackageName()`	Возвращает имя пакета, частью которого является тип класса вызывающего объекта
`ProtectionDomain getProtectionDomain()`	Возвращает домен безопасности, ассоциированный с вызывающим объектом
`Class<? super T> getSuperclass()`	Возвращает суперкласс типа, представленного вызывающим объектом. Возвращенным значением будет `null`, если типом является `Object` или не класс
`boolean isInterface()`	Возвращает `true`, если типом, который представлен вызывающим объектом, является интерфейс, или `false` в противном случае
`String toString()`	Возвращает строковое представление типа, который представлен вызывающим объектом или интерфейсом

Методы, определенные в `Class`, часто полезны в ситуациях, когда требуется информация о типе объекта во время выполнения. Как было показано в табл. 19.15, существуют методы, которые позволяют выяснить дополнительную информацию о конкретном классе, такую как его открытые конструкторы, поля и методы. Помимо прочего это важно для функциональности Java Beans, которая обсуждается далее в книге.

В следующей программе демонстрируется работа метода `getClass()`, унаследованного из `Object`, и метода `getSuperclass()` из `Class`:

```
// Получение информации о типе во время выполнения.
class X {
  int a;
  float b;
}

class Y extends X {
  double c;
}
```

```
class RTTI {
  public static void main(String[] args) {
    X x = new X();
    Y y = new Y();
    Class<?> clObj;
    clObj = x.getClass();  // получить ссылку на Class
    System.out.println("x - объект типа " +
                       clObj.getName());
    clObj = y.getClass();  // получить ссылку на Class
    System.out.println("y - объект типа " +
                       clObj.getName());
    clObj = clObj.getSuperclass();
    System.out.println("Суперклассом y является " +
                       clObj.getName());
  }
}
```

Вот вывод, генерируемый программой:

```
x - объект типа X
y - объект типа Y
Суперклассом y является X
```

Начиная с версии JDK 16, класс `Class` включает методы, которые поддерживают записи. Речь идет о методе `getRecordComponents()`, получающем информацию о компонентах записи, и методе `isRecord()`, который возвращает `true`, если вызывающий объект `Class` представляет запись. Начиная с JDK 17, в `Class` имеются методы, имеющие дело с запечатанными классами и интерфейсы. Это метод `isSealed()`, возвращающий `true`, если вызывающий класс является запечатанным, и метод `getPermittedSubclasses()`, который получает массив экземпляров класса подклассов или подинтерфейсов, разрешенных вызывающим объектом `Class`. Записи, запечатанные классы и интерфейсы обсуждались в главе 17.

Прежде чем двигаться дальше, имеет смысл упомянуть о еще одной возможности класса `Class`, которая может показаться интересной. Начиная с версии JDK 11, в `Class` предлагаются три метода, имеющие отношение к вложению. *Вложение* (nest) — это группа классов и/или интерфейсов, вложенных внутрь внешнего класса или интерфейса. Концепция вложения позволяет машине JVM более эффективно обрабатывать определенные ситуации, связанные с доступом между членами вложения. Важно отметить, что вложение *не* является механизмом исходного кода и *не* меняет язык Java или то, как он определяет доступность. Вложения относятся конкретно к тому, каким образом работают компилятор и машина JVM. Тем не менее, теперь с помощью метода `getNestHost()` можно получить класс/интерфейс верхнего уровня вложения, который называется *хостом вложения*. Метод `isNestMateOf()` позволяет выяснить, является ли один класс/интерфейс членом того же вложения, что и другой. Наконец, вызвав метод `getNestMembers()`, можно получить массив, содержащий список членов вложения. Вы можете счесть упомянутые методы полезными, скажем, при использовании рефлексии.

ClassLoader

Абстрактный класс ClassLoader определяет, как загружаются классы. В приложении можно создавать подклассы, которые расширяют ClassLoader, реализуя его методы, что позволит загружать классы каким-либо способом, отличающимся от того, которым они загружаются исполняющей средой Java в типовых обстоятельствах. Однако обычно поступать подобным образом не придется.

Math

Класс Math содержит методы для всех функций с плавающей точкой, которые применяются в геометрии и тригонометрии, а также несколько методов общего назначения. В Math определены две константы типа double: E (приблизительно 2,72) и PI (приблизительно 3,14).

Тригонометрические функции

В табл. 19.16 кратко описаны методы класса Math, которые принимают параметр double для угла в радианах и возвращают результаты соответствующих тригонометрических функций.

Таблица 19.16. Методы для прямых тригонометрических функций, определенные в классе Math

Метод	Описание
`static double sin (double arg)`	Возвращает синус угла, указанного с помощью arg в радианах
`static double cos (double arg)`	Возвращает косинус угла, указанного с помощью arg в радианах
`static double tan (double arg)`	Возвращает тангенс угла, указанного с помощью arg в радианах

В табл. 19.17 кратко описаны методы класса Math, которые принимают в качестве параметра результат тригонометрической функции и возвращают угол в радианах, приводящий к такому результату. По существу они противоположны методам из табл. 19.16.

Таблица 19.17. Методы для обратных тригонометрических функций, определенные в классе Math

Метод	Описание
`static double asin (double arg)`	Возвращает угол, синус которого указан в arg
`static double acos(double arg)`	Возвращает угол, косинус которого указан в arg
`static double atan(double arg)`	Возвращает угол, тангенс которого указан в arg
`static double atan2 (double x, double y)`	Возвращает угол, тангенс которого равен x/y

В табл. 19.18 кратко описаны методы класса Math, которые вычисляют гиперболический синус, косинус и тангенс угла.

Таблица 19.18. Методы для гиперболических функций, определенные в классе Math

Метод	Описание
`static double sinh(double arg)`	Возвращает гиперболический синус угла, указанного в arg
`static double cosh(double arg)`	Возвращает гиперболический косинус угла, указанного в arg
`static double tanh(double arg)`	Возвращает гиперболический тангенс угла, указанного в arg

Экспоненциальные функции

В табл. 19.19 кратко описаны методы класса Math, которые вычисляют экспоненциальные функции.

Таблица 19.19. Методы для экспоненциальных функций, определенные в классе Math

Метод	Описание
`static double cbrt (double arg)`	Возвращает кубический корень значения arg
`static double exp(double arg)`	Возвращает e в степени arg
`static double expm1(double arg)`	Возвращает e в степени arg-1
`static double log(double arg)`	Возвращает натуральный логарифм значения arg
`static double log10(double arg)`	Возвращает логарифм по основанию 10 значения arg
`static double log1p(double arg)`	Возвращает натуральный логарифм значения arg+1
`static double pow(double y, double x)`	Возвращает y в степени x; например, pow(2.0, 3.0) дает 8.0
`static double scalb (double arg, int factor)`	Возвращает $arg \times 2^{factor}$
`static float scalb (float arg, int factor)`	Возвращает $arg \times 2^{factor}$
`static double sqrt(double arg)`	Возвращает квадратный корень значения arg

Функции округления

В классе Math определено несколько методов, обеспечивающих выполнение различных видов операций округления, которые кратко описаны в табл. 19.20. Обратите внимание на два метода ulp() в конце таблицы. В данном контексте показатель "ulp" означает "units in the last place", т.е. количество *единиц на последнем месте*. Он указывает расстояние между значением и ближайшим более высоким значением и может использоваться для оценки точности результата.

Таблица 19.20. Методы для функций округления, определенные в классе Math

Метод	Описание
`static int abs(int arg)`	Возвращает абсолютное значение arg
`static long abs (long arg)`	Возвращает абсолютное значение arg
`static float abs (float arg)`	Возвращает абсолютное значение arg
`static double abs (double arg)`	Возвращает абсолютное значение arg
`static double ceil (double arg)`	Возвращает наименьшее целое число, которое больше или равно arg
`static double floor(double arg)`	Возвращает наибольшее целое число, которое меньше или равно arg
`static int floorDiv (int dividend, int divisor)`	Возвращает округленный в меньшую сторону результат деления dividend/divisor
`static long floorDiv(long dividend, int divisor)`	Возвращает округленный в меньшую сторону результат деления dividend/divisor
`static long floorDiv(long dividend, long divisor)`	Возвращает округленный в меньшую сторону результат деления dividend/divisor
`static int floorMod(int dividend, int divisor)`	Возвращает округленный в меньшую сторону остаток от деления dividend/divisor
`static int floorMod (long dividend, int divisor)`	Возвращает округленный в меньшую сторону остаток от деления dividend/divisor
`static long floorMod(long dividend, long divisor)`	Возвращает округленный в меньшую сторону остаток от деления dividend/divisor
`static int max (int x, int y)`	Возвращает большее значение среди x и y
`static long max (long x, long y)`	Возвращает большее значение среди x и y

Окончание табл. 19.20

Метод	Описание
`static float max (float x, float y)`	Возвращает большее значение среди x и y
`static double max (double x, double y)`	Возвращает большее значение среди x и y
`static int min(int x, int y)`	Возвращает меньшее значение среди x и y
`static long min (long x, long y)`	Возвращает меньшее значение среди x и y
`static float min (float x, float y)`	Возвращает меньшее значение среди x и y
`static double min (double x, double y)`	Возвращает меньшее значение среди x и y
`static double nextAfter(double arg, double toward)`	Начиная со значения arg, возвращает следующее значение в направлении после toward. Если значения arg и toward равны, тогда возвращается toward
`static float nextAfter (float arg, double toward)`	Начиная со значения arg, возвращает следующее значение в направлении после toward. Если значения arg и toward равны, тогда возвращается toward
`static double nextDown(double val)`	Возвращает следующее значение, которое меньше val
`static float nextDown(float val)`	Возвращает следующее значение, которое меньше val
`static double nextUp(double arg)`	Возвращает следующее значение в положительном направлении от arg
`static float nextUp (float arg)`	Возвращает следующее значение в положительном направлении от arg
`static double rint (double arg)`	Возвращает целое число, ближайшее по значению к arg
`static int round (float arg)`	Возвращает результат округления arg до ближайшего числа типа int
`static long round (double arg)`	Возвращает результат округления arg до ближайшего числа типа long
`static float ulp (float arg)`	Возвращает количество единиц на последнем месте для arg
`static double ulp (double arg)`	Возвращает количество единиц на последнем месте для arg

Прочие методы Math

Помимо показанных выше методов в классе Math определены другие методов, которые кратко описаны в табл. 19.21. Обратите внимание, что в именах некоторых методов присутствует суффикс Exact. Такие методы генерируют исключение ArithmeticException, когда возникает переполнение, что позволяет легко отслеживать различные операции на предмет переполнения.

Таблица 19.21. Прочие методы, определенные в классе Math

Метод	Описание
`static int absExact (int arg)`	Возвращает абсолютное значение arg
`static long absExact (long arg)`	Возвращает абсолютное значение arg
`static int addExact (int arg1, int arg2)`	Возвращает результат сложения arg1 и arg2. При возникновении переполнения генерирует исключение ArithmeticException
`static long addExact (long arg1, long arg2`	Возвращает результат сложения arg1 и arg2. При возникновении переполнения генерирует исключение ArithmeticException
`static double copySign (double arg, double signarg)`	Возвращает значение arg с таким же знаком, как у signarg
`static float copySign (float arg, float signarg)`	Возвращает значение arg с таким же знаком, как у signarg
`static int decrementExact (int arg)`	Возвращает значение arg-1. При возникновении переполнения генерирует исключение ArithmeticException
`static long decrementExact (long arg)`	Возвращает значение arg-1. При возникновении переполнения генерирует исключение ArithmeticException
`static double fma (double arg1, double arg2, double arg3)`	Добавляет arg3 к произведению arg1 и arg2 и возвращает округленный результат. Имя метода является сокращением от "fused multiply add" — комбинированное умножение и сложение
`static float fma (float arg1, float arg2, float arg3)`	Добавляет arg3 к произведению arg1 и arg2 и возвращает округленный результат
`static int getExponent (double arg)`	Возвращает показатель степени 2, используемый двоичным представлением arg

Продолжение табл. 19.21

Метод	Описание
`static int getExponent(float arg)`	Возвращает показатель степени 2, используемый двоичным представлением arg
`static hypot(double side1, double side2)`	Возвращает длину гипотенузы прямоугольного треугольника по длине двух катетов
`static double IEEEremainder(double dividend, double divisor)`	Возвращает остаток от деления dividend/divisor
`static int incrementExact (int arg)`	Возвращает значение arg+1. При возникновении переполнения генерирует исключение ArithmeticException
`static long incrementExact (long arg)`	Возвращает значение arg+1. При возникновении переполнения генерирует исключение ArithmeticException
`static int multiplyExact (int arg1, int arg2)`	Возвращает значение arg1*arg2. При возникновении переполнения генерирует исключение ArithmeticException
`static long multiplyExact (long arg1, int arg2)`	Возвращает значение arg1*arg2. При возникновении переполнения генерирует исключение ArithmeticException
`static long multiplyExact (long arg1, long arg2)`	Возвращает значение arg1*arg2. При возникновении переполнения генерирует исключение ArithmeticException
`static long multiplyFull (int arg1, int arg2)`	Возвращает значение arg1*arg2 как величину типа long
`static long multiplyHigh (long arg1, long arg2)`	Возвращает значение типа long, которое содержит старшие биты значения arg1*arg2
`static int negateExact (int arg)`	Возвращает значение -arg. При возникновении переполнения генерирует исключение ArithmeticException
`static long negateExact (long arg)`	Возвращает значение -arg. При возникновении переполнения генерирует исключение ArithmeticException
`static double random()`	Возвращает псевдослучайное число между 0 и 1

Окончание табл. 19.21

Метод	Описание
`static float signum (double arg)`	Определяет знак значения. Возвращает 0, если arg равно 0; 1, если arg больше 0; –1, если arg меньше 0
`static float signum (float arg)`	Определяет знак значения. Возвращает 0, если arg равно 0; 1, если arg больше 0; –1, если arg меньше 0
`static int subtractExact (int arg1, int arg2)`	Возвращает значение arg1–arg2. При возникновении переполнения генерирует исключение ArithmeticException
`static long subtractExact (long arg1, long arg2)`	Возвращает значение arg1–arg2. При возникновении переполнения генерирует исключение ArithmeticException
`static double toDegrees (double angle)`	Преобразует радианы в градусы. Передаваемый в angle угол должен быть указан в радианах. Возвращается результат в градусах
`static int toIntExact (long arg)`	Возвращает значение arg как величину типа int. При возникновении переполнения генерирует исключение ArithmeticException
`static double toRadians (double angle)`	Преобразует градусы в радианы. Передаваемый в angle угол должен быть указан в градусах. Возвращается результат в радианах

В следующей программе демонстрируется работа методов `toRadians()` и `toDegrees()`:

```
// Демонстрация работы toDegrees() и toRadians().
class Angles {
  public static void main(String[] args) {
    double theta = 120.0;

    System.out.println(theta + " градусов равно " +
                       Math.toRadians(theta) + " радиан.");

    theta = 1.312;
    System.out.println(theta + " радиан равно " +
                       Math.toDegrees(theta) + " градусов.");
  }
}
```

Ниже показан вывод:

```
120.0 градусов равно 2.0943951023931953 радиан.
1.312 радиан равно 75.17206272116401 градусов.
```

StrictMath

В классе StrictMath определен полный набор математических методов, которые аналогичны методам класса Math. Разница в том, что методы из StrictMath гарантированно генерируют идентичные результаты во всех реализациях Java, тогда как методы из Math имеют большую свободу действий в целях повышения производительности. Важно отметить, что после выхода версии JDK 17 все математические вычисления выполняются строго.

Compiler

Класс Compiler поддерживает создание сред Java, в которых байт-код Java компилируется в исполняемый код, а не интерпретируется. Он не предназначен для обычного применения в программировании, к тому же он помечен как устаревший и подлежащий удалению.

Thread, ThreadGroup и Runnable

Интерфейс Runnable вместе с классами Thread и ThreadGroup поддерживают многопоточное программирование. Все они исследуются далее.

На заметку! Обзор приемов, используемых для управления потоками, реализации интерфейса Runnable и создания многопоточных программ, был представлен в главе 11.

Интерфейс Runnable

Интерфейс Runnable должен быть реализован любым классом, который будет инициировать отдельный поток выполнения. В Runnable определен только один абстрактный метод по имени run(), являющийся точкой входа в поток:

```
void run()
```

Создаваемые потоки обязаны реализовывать данный метод.

Класс Thread

Класс Thread создает новый поток выполнения. Он реализует интерфейс Runnable и определяет несколько конструкторов, часть которых показана ниже:

```
Thread()
Thread(Runnable threadOb)
Thread(Runnable threadOb, String threadName)
Thread(String threadName)
Thread(ThreadGroup groupOb, Runnable threadOb)
Thread(ThreadGroup groupOb, Runnable threadOb, String threadName)
Thread(ThreadGroup groupOb, String threadName)
```

Здесь threadOb — это экземпляр класса, который реализует интерфейс Runnable и определяет, где начнется выполнение потока. Имя потока определяется параметром threadName. Когда имя не указано, оно создается виртуальной машиной Java. В groupOb задается группа потоков, к которой будет принадлежать новый поток. Если группа потоков не указана, то по умолчанию новый поток принадлежит той же группе, что и родительский поток. В классе Thread определены следующие константы:

```
MAX_PRIORITY
MIN_PRIORITY
NORM_PRIORITY
```

Приведенные константы вполне ожидаемо указывают максимальный, минимальный и стандартный приоритеты потоков. В табл. 19.22 кратко описаны методы класса Thread, не объявленные нерекомендуемыми. В состав Thread также входят нерекомендуемые методы stop(), suspend() и resume(). Тем не менее, как объяснялось в главе 11, они были объявлены нерекомендуемыми из-за присущей им нестабильности. Кроме того, нерекомендуемым является метод countStackFrames(), потому что он вызывает suspend(), и метод destroy(), поскольку он способен приводить к взаимоблокировке. Наконец, начиная с версии JDK 11, метод destroy() и одна версия метода stop() удалены из Thread, а в версии JDK 17 метод checkAccess() помечен как устаревший и подлежащий удалению.

Таблица 19.22. Методы класса Thread, которые не объявлены нерекомендуемыми

Метод	Описание
`static int activeCount()`	Возвращает приблизительное число активных потоков в группе, которой принадлежит данный поток
`static Thread currentThread()`	Возвращает объект Thread, который инкапсулирует поток, вызывающий этот метод
`static void dumpStack()`	Отображает стек вызовов для потока
`static int enumerate (Thread[] threads)`	Помещает копии всех объектов Thread из группы текущего потока в threads и возвращает количество потоков
`static Map<Thread, StackTraceElement[]> getAllStackTraces()`	Возвращает объект Map, содержащий трассировки стека для всех активных потоков. Каждый элемент Map состоит из ключа, который является объектом Thread, и его значения, представляющего собой массив StackTraceElement
`ClassLoader getContextClassLoader()`	Возвращает загрузчик классов контекста, который используется для загрузки классов и ресурсов для данного потока

Продолжение табл. 19.22

Метод	Описание
`static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()`	Возвращает стандартный обработчик неперехваченных исключений
`long getID()`	Возвращает идентификатор вызывающего потока
`final String getName()`	Возвращает имя потока
`final int getPriority()`	Возвращает настройку приоритета потока
`StackTraceElement[] getStackTrace()`	Возвращает массив, содержащий трассировку стека для вызывающего потока
`Thread.State getState()`	Возвращает состояние вызывающего потока
`final ThreadGroup getThreadGroup()`	Возвращает объект ThreadGroup, членом которого является вызывающий поток
`Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()`	Возвращает обработчик неперехваченных исключений вызывающего потока
`static boolean holdsLock (Object ob)`	Возвращает true, если вызывающий поток владеет блокировкой на ob, или false в противном случае
`void interrupt()`	Прерывает поток
`static boolean interrupted()`	Возвращает true, если выполняющийся в текущий момент поток был прерван, или false в противном случае
`final boolean isAlive()`	Возвращает true, если поток все еще активен, или false в противном случае
`final boolean isDaemon()`	Возвращает true, если поток является потоком демона, или false в противном случае
`boolean isInterrupted()`	Возвращает true, если вызывающий поток был прерван, или false в противном случае
`final void join() throws InterruptedException`	Организует ожидание завершения потока
`final void join (long milliseconds) throws InterruptedException`	Организует ожидание завершения потока, на котором был вызван, в течение указанного в milliseconds времени в миллисекундах

Окончание табл. 19.22

Метод	Описание
`final void join(long milliseconds, int nanoseconds) throws InterruptedException`	Организует ожидание завершения потока, на котором был вызван, в течение указанного в `milliseconds` и `nanoseconds` времени в миллисекундах и наносекундах
`static void onSpinWait()`	Вызывается для объявления о том, что поток в текущий момент находится внутри цикла ожидания, делая возможной оптимизацию во время выполнения
`void run()`	Начинает выполнение потока
`void setContextClassLoader(ClassLoader cl)`	Устанавливает загрузчик классов контекста, который будет использоваться вызывающим потоком, в `cl`
`final void setDaemon(boolean state)`	Помечает поток как поток демона
`static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler e)`	Устанавливает стандартный обработчик неперехваченных исключений в `e`
`final void setName(String threadName)`	Устанавливает имя потока в `threadName`
`final void setPriority(int priority)`	Устанавливает приоритет потока в `priority`
`void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler e)`	Устанавливает стандартный обработчик неперехваченных исключений вызывающего потока в `e`
`static void sleep(long milliseconds) throws InterruptedException`	Приостанавливает выполнение потока на указанное в `milliseconds` количество миллисекунд
`static void sleep(long milliseconds, int nanoseconds) throws InterruptedException`	Приостанавливает выполнение потока на указанное в `milliseconds` количество миллисекунд плюс заданное в `nanoseconds` количество наносекунд
`void start()`	Начинает выполнение потока
`String toString()`	Возвращает строковый эквивалент потока
`static void yield()`	Обеспечивает то, что вызывающий поток предлагает уступить ЦП другому потоку

Класс ThreadGroup

Класс ThreadGroup создает группу потоков. В нем определены два конструктора:

```
ThreadGroup(String groupName)
ThreadGroup(ThreadGroup parentOb, String groupName)
```

Для обеих форм в groupName указывается имя группы потока. Первая версия конструктора создает новую группу с текущим потоком в качестве родительского. Во второй версии конструктора родительский поток задается с помощью parentOb. В табл. 19.23 кратко описаны методы ThreadGroup, не объявленные нерекомендуемыми.

Таблица 19.23. Методы класса ThreadGroup, которые не объявлены нерекомендуемыми

Метод	Описание
`int activeCount()`	Возвращает приблизительное число активных потоков в вызывающей группе (включая активные потоки в подгруппах)
`int activeGroupCount()`	Возвращает приблизительное число активных групп (включая подгруппы), для которых вызывающий поток является родительским
`int enumerate (Thread[] group)`	Помещает активные потоки, входящие в группу вызывающего потока (включая активные потоки в подгруппах), в массив group
`int enumerate (Thread[] group, boolean all)`	Помещает активные потоки, которые входят в группу вызывающего потока, в массив group. Если all равно true, то в group также помещаются потоки во всех подгруппах потока
`int enumerate (ThreadGroup[] group)`	Помещает активные подгруппы (включая подгруппы подгрупп и т.д.) группы вызывающего потока в массив group
`int enumerate (ThreadGroup[] group, boolean all)`	Помещает активные подгруппы группы вызывающего потока в массив group. Если all равно true, то в group также помещаются все активные подгруппы подгрупп (и т.д.)
`final int getMaxPriority()`	Возвращает настройку максимального приоритета для группы
`final String getName()`	Возвращает имя группы

Окончание табл. 19.23

Метод	Описание
`final ThreadGroup getParent()`	Возвращает `null`, если вызывающий объект `ThreadGroup` не имеет родителя. В противном случае возвращает родителя вызывающего объекта
`final void interrupt()`	Вызывает метод `interrupt()` всех потоков в группе и любых подгруппах
`void list()`	Отображает информацию о группе
`final Boolean parentOf (ThreadGroup group)`	Возвращает `true`, если вызывающий поток является родительским для `group` (либо самой группой `group`), или `false` в противном случае
`final void setMaxPriority (int priority)`	Устанавливает максимальный приоритет вызывающей группы в `priority`
`String toString()`	Возвращает строковый эквивалент группы
`void uncaughtException (Thread thread, Throwable e)`	Вызывается, когда исключение становится неперехваченным

Группы потоков предлагают удобный способ управления группами потоков как единым целым. Это особенно полезно в ситуациях, когда желательно приостановить и возобновить несколько связанных потоков. Например, представьте себе программу, в которой один набор потоков используется для печати документа, другой набор применяется для вывода документа на экран и еще один набор сохраняет документ в дисковом файле. Если печать прерывается, то нужен простой способ останова всех потоков, связанных с печатью. Группы потоков позволяют эффективно решить задачу. В следующей программе, создающей две группы потоков по два потока в каждой, иллюстрируется такое использование:

```
// Демонстрация использования групп потоков.
class NewThread extends Thread {
  boolean suspendFlag;

  NewThread(String threadname, ThreadGroup tgOb) {
    super(tgOb, threadname);
    System.out.println("Новый поток: " + this);
    suspendFlag = false;
  }

  // Точка входа для потока.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(getName() + ": " + i);
```

```
        Thread.sleep(1000);
        synchronized(this) {
          while(suspendFlag) {
            wait();
          }
        }
      }
    } catch (Exception e) {
      System.out.println("Исключение в " + getName());
    }
    System.out.println(getName() + " завершает работу.");
  }
  synchronized void mysuspend() {
    suspendFlag = true;
  }
  synchronized void myresume() {
    suspendFlag = false;
    notify();
  }
}
class ThreadGroupDemo {
  public static void main(String[] args) {
    ThreadGroup groupA = new ThreadGroup("Group A");
    ThreadGroup groupB = new ThreadGroup("Group B");

    NewThread ob1 = new NewThread("One", groupA);
    NewThread ob2 = new NewThread("Two", groupA);
    NewThread ob3 = new NewThread("Three", groupB);
    NewThread ob4 = new NewThread("Four", groupB);

    ob1.start();
    ob2.start();
    ob3.start();
    ob4.start();

    System.out.println("\nВывод из list():");
    groupA.list();
    groupB.list();
    System.out.println();

    System.out.println("Приостановка Group A");
    Thread[] tga = new Thread[groupA.activeCount()];
    groupA.enumerate(tga);  // получить потоки в группе
    for(int i = 0; i < tga.length; i++) {
     ((NewThread)tga[i]).mysuspend(); //приостановить работу каждого потока
    }

    try {
      Thread.sleep(4000);
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван.");
    }

    System.out.println("Возобновление Group A");
    for(int i = 0; i < tga.length; i++) {
     ((NewThread)tga[i]).myresume(); //возобновить работу потоков в группе
    }
```

```
    // Ожидать завершения работы потоков.
    try {
      System.out.println("Ожидание завершения работы потоков.");
      ob1.join();
      ob2.join();
      ob3.join();
      ob4.join();
    } catch (Exception e) {
      System.out.println("Исключение в главном потоке");
    }
    System.out.println("Главный поток завершает работу.");
  }
}
```

Ниже показан пример вывода, выдаваемого программой (вывод у вас может отличаться):

```
Новый поток: Thread[One,5,Group A]
Новый поток: Thread[Two,5,Group A]
Новый поток: Thread[Three,5,Group B]
Новый поток: Thread[Four,5,Group B]
Вывод из list():
java.lang.ThreadGroup[name=Group A,maxpri=10]
  Thread[One,5,Group A]
  Thread[Two,5,Group A]
java.lang.ThreadGroup[name=Group B,maxpri=10]
  Thread[Three,5,Group B]
  Thread[Four,5,Group B]
Приостановка Group A
Three: 5
Four: 5
Three: 4
Four: 4
Three: 3
Four: 3
Three: 2
Four: 2
Возобновление Group A
Ожидание завершения работы потоков.
One: 5
Two: 5
Three: 1
Four: 1
One: 4
Two: 4
Three завершает работу.
Four завершает работу.
One: 3
Two: 3
One: 2
Two: 2
One: 1
Two: 1
One завершает работу.
Two завершает работу.
Главный поток завершает работу.
```

Обратите внимание, что внутри программы группа потоков Group A приостанавливается на четыре секунды. Как подтверждает вывод, это приводит к приостановке потоков One и Two, но потоки Three и Four продолжают выполняться. По истечении четырех секунд потоки One и Two возобновляют работу. Обратите внимание на то, каким образом приостанавливает и возобновляет работу группа потоков Group A. Сначала из группы Group A получаются потоки за счет вызова для нее метода enumerate(). Затем каждый поток приостанавливается в ходе итерации по результирующему массиву. Для возобновления работы потоков в Group A массив снова просматривается, и каждый поток возобновляется.

ThreadLocal и InheritableThreadLocal

В пакете java.lang определены два дополнительных класса, связанные с потоками:

- **ThreadLocal**. Применяется для создания локальных переменных потока. Каждый поток будет иметь собственную копию локальной переменной потока.
- **InheritableThreadLocal**. Используется для создания локальных переменных потока, которые могут быть унаследованы.

Package

Класс Package инкапсулирует информацию о пакете. Методы, определенные в Package, кратко описаны в табл. 19.24.

Таблица 19.24. Методы, определенные в классе Package

Метод	Описание
`<A extends Annotation> A getAnnotation (Class<A> annoType)`	Возвращает объект Annotation, содержащий аннотацию, которая ассоциирована с вызывающим объектом и имеет тип annoType
`Annotation[] getAnnotations()`	Возвращает все аннотации, ассоциированные с вызывающим объектом, в массиве объектов Annotation. Возвращает ссылку на результирующий массив
`<A extends Annotation> A[] getAnnotationsByType (Class<A> annoType)`	Возвращает массив аннотаций (включающий повторяющиеся аннотации) типа annoType, которые ассоциированы с вызывающим объектом

Продолжение табл. 19.24

Метод	Описание
`<A extends Annotation> A getDeclaredAnnotation (Class<A> annoType)`	Возвращает объект Annotation, содержащий неунаследованную аннотацию, которая ассоциирована с вызывающим объектом и имеет тип annoType
`Annotation[] getDeclaredAnnotations()`	Возвращает объекты Annotation для всех аннотаций, которые объявляются вызывающим объектом. (Унаследованные аннотации игнорируются.)
`<A extends Annotation> A[] getDeclaredAnnotationsByType (Class<A> annoType)`	Возвращает массив неунаследованных аннотаций (включающий повторяющиеся аннотации) типа annoType, которые ассоциированы с вызывающим объектом
`String getImplementationTitle()`	Возвращает заголовок вызывающего пакета
`String getImplementationVendor()`	Возвращает имя поставщика реализации вызывающего пакета
`String getImplementationVersion()`	Возвращает номер версии вызывающего пакета
`String getName()`	Возвращает имя вызывающего пакета
`static Package[] getPackages()`	Возвращает все пакеты, о которых в текущий момент осведомлена вызывающая программа
`String getSpecificationTitle()`	Возвращает заголовок спецификации вызывающего пакета
`String getSpecificationVendor()`	Возвращает имя владельца спецификации для вызывающего пакета
`String getSpecificationVersion()`	Возвращает номер версии спецификации вызывающего пакета
`int hashCode()`	Возвращает хеш-код для вызывающего пакета
`boolean isAnnotationPresent (Class<? extends Annotation> anno)`	Возвращает true, если аннотация, описанная с помощью anno, ассоциирована с вызывающим объектом, или false в противном случае
`boolean isCompatibleWith (String verNum) throws NumberFormatException`	Возвращает true, если значение verNum меньше или равно номеру версии вызывающего пакета

Окончание табл. 19.24

Метод	Описание
`boolean isSealed()`	Возвращает `true`, если вызывающий пакет является запечатанным, или `false` в противном случае
`boolean isSealed(URL url)`	Возвращает `true`, если вызывающий пакет является запечатанным относительно `url`, или `false` в противном случае
`String toString()`	Возвращает строковый эквивалент вызывающего пакета

В показанной далее программе демонстрируется применение класса `Package` для отображения пакетов, о которых программа осведомлена в текущий момент:

```
// Демонстрация использования Package.
class PkgTest {
  public static void main(String[] args) {
    Package[] pkgs;

    pkgs = Package.getPackages();

    for(int i=0; i < pkgs.length; i++)
      System.out.println(
        pkgs[i].getName() + " " +
        pkgs[i].getImplementationTitle() + " " +
        pkgs[i].getImplementationVendor() + " " +
        pkgs[i].getImplementationVersion()
      );
  }
}
```

Module

Класс `Module`, появившийся в JDK 9, инкапсулирует модуль. С помощью экземпляра `Module` можно добавлять различные права доступа к модулю, определять права доступа или получать информацию о модуле. Например, чтобы экспортировать пакет в указанный модуль, вызовите `addExports()`; чтобы открыть пакет для указанного модуля, вызовите `addOpens()`; чтобы прочитать другой модуль, вызовите `addReads()`; чтобы затребовать службу, вызовите `addUses()`. Для определения, может ли модуль иметь доступ к другому модулю, вызовите `canRead()`. Чтобы определить, использует ли модуль службу, вызовите `canUse()`. В то время как упомянутые методы будут полезнее всего в особых ситуациях, в классе `Module` определено несколько других методов, которые могут представлять более общий интерес.

Например, получить имя модуля можно посредством вызова метода `getName()`, который для именованного модуля возвращает имя, а для неиме-

нованного — null. Получить набор пакетов в модуле можно, вызвав метод getPackages(), который возвращает дескриптор модуля в виде экземпляра ModuleDescriptor. (ModuleDescriptor — это класс, объявленный в пакете java.lang.module.) Определить, пакет экспортируется или открывается вызывающим модулем, можно с помощью вызова isExported() или isOpen(). Для выяснения, модуль был именован или нет, применяйте метод isNamed(). Другие методы включают getAnnotation(), getDeclaredAnnotations(), getLayer(), getClassLoader() и getResourceAsStream(). Кроме того, в Module переопределяется метод toString().

При наличии модулей, определенных внутри примеров в главе 16, вы можете легко поэкспериментировать с классом Module. Например, добавьте в класс MyModAppDemo следующие строки:

```
Module myMod = MyModAppDemo.class.getModule();
System.out.println("Модуль: " + myMod.getName());

System.out.print("Пакеты: ");
for(String pkg : myMod.getPackages()) System.out.println(pkg + " ");
```

В примере используются методы getName() и getPackages(). Обратите внимание, что экземпляр модуля получается вызовом getModule() на экземпляре Class для MyModAppDemo. В результате запуска эти строки производят такой вывод:

```
Модуль: appstart
Пакеты: appstart.mymodappdemo
```

ModuleLayer

Класс ModuleLayer, появившийся в JDK 9, инкапсулирует уровень модуля. Вложенный класс ModuleLayer.Controller, также введенный в JDK 9, является контроллером уровня модуля. В целом эти классы предназначены для специализированных приложений.

RuntimePermission

Класс RuntimePermission относится к механизму безопасности Java.

Throwable

Класс Throwable поддерживает систему обработки исключений Java и представляет собой класс, от которого унаследованы все классы исключений. Он обсуждался в главе 10.

SecurityManager

Класс SecurityManager в версии JDK 17 был объявлен устаревшим и подлежащим удалению. Последние сведения ищите в документации по Java.

StackTraceElement

Класс StackTraceElement описывает одиночный *стековый фрейм*, который является индивидуальным элементом трассировки стека, когда возникает исключение. Каждый стековый фрейм представляет *точку выполнения*, которая включает в себя имя класса, имя метода, имя файла и номер строки исходного кода. Начиная с JDK 9, также включается информация о модуле. В классе StackTraceElement определены два конструктора, но обычно необходимость в их применении не возникает, поскольку массив StackTraceElements возвращается различными методами, такими как метод getStackTrace() в классах Throwable и Thread.

В табл. 19.25 кратко описаны методы, поддерживаемые StackTraceElement; они предоставляют программный доступ к трассировке стека.

Таблица 19.25. Методы, определенные в классе StackTraceElement

Метод	Описание
boolean equals (Object ob)	Возвращает true, если вызывающий объект StackTraceElement является тем же, что и переданный в ob, или false в противном случае
String getClassLoaderName()	Возвращает имя загрузчика классов, используемого для загрузки класса, в котором оказалась точка выполнения, описанная вызываемым объектом StackTraceElement. Если объект не включает информацию о загрузчике классов, тогда возвращается null
String getClassName()	Возвращает имя класса, в котором оказалась точка выполнения, описанная вызываемым объектом StackTraceElement
String getFileName()	Возвращает имя файла, в котором хранится исходный код точки выполнения, описанный вызываемым объектом StackTraceElement
int getLineNumber()	Возвращает номер строки исходного кода, в которой оказалась точка выполнения, описанная вызываемым объектом StackTraceElement. В ряде ситуаций номер строки не будет доступным, и тогда возвращается отрицательное значение
String getMethodName()	Возвращает имя метода, в котором оказалась точка выполнения, описанная вызывающим объектом StackTraceElement
String getModuleName()	Возвращает имя модуля, в котором оказалась точка выполнения, описанная вызывающим объектом StackTraceElement. Если объект не содержит информацию о модуле, тогда возвращается null

Окончание табл. 19.25

Метод	Описание
`String getModuleVersion()`	Возвращает версию модуля, в котором оказалась точка выполнения, описанная вызывающим объектом `StackTraceElement`. Если объект не содержит информацию о модуле, тогда возвращается `null`
`int hashCode()`	Возвращает хеш-код для вызывающего объекта `StackTraceElement`
`boolean isNativeMethod()`	Возвращает `true`, если точка выполнения, описанная вызывающим объектом `StackTraceElement`, оказалась в собственном методе, или `false` в противном случае
`String toString()`	Возвращает строковый эквивалент вызывающей последовательности

StackWalker и StackWalker.StackFrame

Добавленный в версии JDK 9 класс `StackWalker` и интерфейс `StackWalker.StackFrame` поддерживают операции прохода по стеку. Экземпляр `StackWalker` получается с помощью статического метода `getInstance()`, определенного в `StackWalker`. Проход по стеку инициируется вызовом метода `walk()` из `StackWalker`. Каждый стековый фрейм инкапсулируется в объекте `StackWalker.StackFrame`. Кроме того, появилось перечисление `StackWalker.Option`.

Enum

Как было описано в главе 12, перечисление представляет собой список именованных констант. (Вспомните, что перечисление создается с использованием ключевого слова `enum`.) Все перечисления автоматически наследуются от `Enum` — обобщенного класса, объявленного следующим образом:

```
class Enum<E extends Enum<E>>
```

Здесь `E` обозначает тип перечисления. Класс `Enum` не имеет открытых конструкторов.

В табл. 19.26 кратко описаны часто применяемые методы класса `Enum`. Начиная с версии JDK 12, класс `Enum` также реализует интерфейс `Constable`, в котором определен метод `describeConstable()`. В версии JDK 12 был добавлен класс `Enum.EnumDesc`.

Таблица 19.26. Часто используемые методы, определенные в классе Enum

Метод	Описание
`protected final Object clone() throws CloneNotSupportedException`	Вызов этого метода приводит к генерации исключения `CloneNotSupportedException`, что предотвращает клонирование перечислений
`final int compareTo(E e)`	Сравнивает порядковые значения двух констант одного перечисления. Возвращает отрицательное значение, если порядковое значение вызывающей константы меньше e, ноль, если два порядковых значения совпадают, или положительное значение, если порядковое значение вызывающей константы больше e
`final boolean equals (Object obj)`	Возвращает `true`, если `obj` и вызывающий объект ссылаются на одну и ту же константу
`final Class<E> getDeclaringClass()`	Возвращает тип перечисления, членом которого является вызывающая константа
`final int hashCode()`	Возвращает хеш-код для вызывающего объекта
`final String name()`	Возвращает неизмененное имя вызывающей константы
`final int ordinal()`	Возвращает значение, которое указывает позицию константы перечисления в списке констант
`String toString()`	Возвращает имя вызывающей константы, которое может отличаться от имени, используемого в объявлении перечисления
`static <T extends Enum<T>> T valueOf(Class<T> e-type, String name)`	Возвращает константу, которая ассоциирована с именем в типе перечисления, указанном в `e-type`

Record

Класс `Record`, добавленный в версии JDK 16, является суперклассом для всех записей. Другими словами, все записи автоматически наследуются от `Record`. В нем не определяются собственные методы, но переопределяются методы `equals()`, `hashCode()` и `toString()`, унаследованные от `Object`. Записи обсуждались в главе 17.

ClassValue

Класс ClassValue можно использовать для ассоциирования значения с типом. Это обобщенный класс, который определен, как показано ниже:

```
Class ClassValue<T>
```

Класс ClassValue предназначен для узкоспециализированных целей, а не для обычного программирования.

Интерфейс CharSequence

В интерфейсе CharSequence определены методы, предоставляющие доступ только для чтения к последовательности символов. Они кратко описаны в табл. 19.27. Интерфейс CharSequence реализуется помимо прочих классами String, StringBuffer и StringBuilder.

Таблица 19.27. Методы, определенные в классе CharSequence

[illegible]	[illegible]
`char charAt(int idx)`	Возвращает символ по индексу, указанному в idx
`static int compare(CharSequence seqA, CharSequence seqB)`	Сравнивает seqA и seqB. Возвращает ноль, если последовательности одинаковы, отрицательное значение, если seqA меньше seqB, или положительное значение, если seqA больше seqB
`default IntStream chars()`	Возвращает поток данных (в виде экземпляра IntStream) для символов в вызывающем объекте
`default IntStream codePoints()`	Возвращает поток данных (в виде экземпляра IntStream) для кодовых точек в вызывающем объекте
`default boolean isEmpty()`	Возвращает true, если вызывающая последовательность не содержит символов, или false в противном случае
`int length()`	Возвращает количество символов в вызывающей последовательности
`CharSequence subSequence(int startIdx, int stopIdx)`	Возвращает подмножество вызывающей последовательности, начиная с индекса startIdx и заканчивая индексом stopIdx-1
`String toString()`	Возвращает строковый эквивалент вызывающей последовательности

Интерфейс Comparable

Объекты классов, реализующих интерфейс Comparable, можно упорядочивать. Другими словами, реализующие Comparable классы содержат объекты, которые можно сравнивать каким-то осмысленным образом. Интерфейс Comparable является обобщенным и объявлен следующим образом:

```
interface Comparable<T>
```

Здесь T представляет тип сравниваемых объектов.

В интерфейсе Comparable объявлен один метод, который применяется для определения того, что Java называет *естественным порядком* следования экземпляров класса. Вот сигнатура этого метода:

```
int compareTo(T obj)
```

Метод compareTo() сравнивает вызываемый объект с obj. Он возвращает ноль, если значения равны, отрицательное значение, если вызывающий объект имеет значение меньше obj, или положительное значение, если вызывающий объект имеет значение больше obj.

Интерфейс Comparable реализуется многими классами, которые уже рассматривались в книге, в том числе Byte, Character, Double, Float, Long, Short, String, Integer и Enum.

Интерфейс Appendable

К объекту класса, реализующего интерфейс Appendable, можно добавлять символ или последовательность символов. В Appendable определены три метода:

```
Appendable append(char ch) throws IOException
Appendable append(CharSequence chars) throws IOException
Appendable append(CharSequence chars, int begin, int end)
   throws IOException
```

В первой форме к вызывающему объекту добавляется символ ch. Во второй форме к вызывающему объекту добавляется последовательность символов chars. Третья форма позволяет указывать часть последовательности chars (символы от begin до end-1). Во всех случаях возвращается ссылка на вызывающий объект.

Интерфейс Iterable

Интерфейс Iterable должен быть реализован любым классом, объекты которого будут задействованы в цикле for стиля "for-each". Другими словами, чтобы объект можно было использовать в цикле for стиля "for-each", его класс должен реализовывать Iterable — обобщенный интерфейс, объявление которого показано ниже:

```
interface Iterable<T>
```

В T указывается тип объектов, участвующих в итерации.

В интерфейсе Iterable определен один абстрактный метод iterator():

```
Iterator<T> iterator()
```

Он возвращает итератор для элементов, которые содержатся в вызывающем объекте.

Кроме того, в интерфейсе Iterable определены два метода со стандартной реализацией. Первый называется forEach():

```
default void forEach(Consumer<? super T> action)
```

Для каждого итерируемого элемента метод forEach() выполняет код, указанный в action. (Тип Consumer — это функциональный интерфейс, определенный в пакете java.util.function, как объясняется в главе 21.)

Второй метод со стандартной реализацией называется spliterator():

```
default Spliterator<T> spliterator()
```

Он возвращает сплитератор для последовательности, участвующей в итерации. (Сплитераторы обсуждаются в главах 20 и 30.)

На заметку! Итераторы подробно рассматриваются в главе 20.

Интерфейс Readable

Интерфейс Readable указывает, что объект можно применять в качестве источника символов. В нем определен один метод по имени read():

```
int read(CharBuffer buf ) throws IOException
```

Метод read() считывает символы в buf. Он возвращает количество прочитанных символов или -1, если был достигнут конец файла.

Интерфейс AutoCloseable

Интерфейс AutoCloseable обеспечивает поддержку оператора try с ресурсами, который реализует то, что иногда называют *автоматическим управлением ресурсами* (automatic resource management — ARM). Оператор try с ресурсами автоматизирует процесс освобождения ресурса (такого как поток), когда он больше не нужен. (Подробные сведения ищите в главе 13.) Использовать в операторе try с ресурсами можно только объекты классов, которые реализуют AutoCloseable. В интерфейсе AutoCloseable определен единственный метод close():

```
void close() throws Exception
```

Метод close() закрывает вызывающий объект, освобождая все ресурсы, которые тот мог удерживать. Он автоматически вызывается в конце оператора try с ресурсами, что устраняет необходимость в явном вызове close(). Интерфейс AutoCloseable реализуется несколькими классами, в том числе всеми классами ввода-вывода, открывающими поток, который можно закрыть.

Интерфейс Thread.UncaughtExceptionHandler

Статический интерфейс Thread.UncaughtExceptionHandler реализуется классами, которым необходимо обрабатывать неперехваченные исключения. Он реализован классом ThreadGroup. В интерфейсе Thread.UncaughtExceptionHandler объявлен только один метод:

```
void uncaughtException(Thread thrd, Throwable exc)
```

Здесь thrd — ссылка на поток, сгенерировавший исключение, а exc — ссылка на исключение.

Подпакеты java.lang

В пакете java.lang определено несколько подпакетов. Пакеты находятся в модуле java.base, если не указано иное.

```
java.lang.annotation
java.lang.constant
java.lang.instrument
java.lang.invoke
java.lang.management
java.lang.module
java.lang.ref
java.lang.reflect
```

Все они кратко описаны ниже.

java.lang.annotation

Средство аннотирования Java поддерживается пакетом java.lang.annotation, в котором определен интерфейс Annotation, перечисления ElementType и RetentionPolicy, а также ряд предопределенных аннотаций. Аннотации обсуждались в главе 12.

java.lang.constant

java.lang.constant — это специализированный пакет, поддерживающий дескрипторы для констант. Обычно он используется приложениями, которые имеют доступ к байт-коду, и появился в JDK 12.

java.lang.instrument

В пакете java.lang.instrument определены средства, которые можно применять для добавления инструментария к различным аспектам выполнения программ, а именно — интерфейсы Instrumentation и ClassFileTransformer и класс ClassDefinition.

Пакет java.lang.instrument находится в модуле java.instrument.

java.lang.invoke

Пакет java.lang.invoke поддерживает динамические средства языка. Он содержит такие классы, как CallSite, MethodHandle и MethodType.

java.lang.management

Пакет java.lang.management обеспечивает поддержку управления для виртуальной машины Java и исполняющей среды. С помощью средств java.lang.management можно наблюдать за различными аспектами выполнения программы и управлять ими. Этот пакет находится в модуле java.management.

java.lang.module

Пакет java.lang.module поддерживает модули. В нем определены классы ModuleDescriptor и ModuleReference, а также интерфейсы ModuleFinder и ModuleReader.

java.lang.ref

Ранее вы узнали, что средства сборки мусора в Java автоматически определяют отсутствие ссылок на объект. Затем выдвигается допущение, что объект больше не нужен, и его память освобождается. Классы в пакете java.lang.ref обеспечивают более гибкое управление процессом сборки мусора.

java.lang.reflect

Рефлексия — это способность программы анализировать код во время выполнения. Пакет java.lang.reflect предоставляет возможность получить информацию о полях, конструкторах, методах и модификаторах класса. Помимо прочих причин такая информация необходима для создания программных инструментов, которые позволяют работать с компонентами Java Beans. Инструменты используют рефлексию для динамического определения характеристик компонента. Рефлексия была представлена в главе 12 и подробно рассматривается в главе 31.

В пакете java.lang.reflect определено несколько классов, в том числе Method, Field и Constructor. В нем также определен ряд интерфейсов, включая AnnotatedElement, Member и Type. Кроме того, в состав пакета java.lang.reflect входит класс Array, позволяющий динамически создавать массивы и получать к ним доступ.

ГЛАВА

20 Пакет java.util, часть 1: Collections Framework

В настоящей главе начинается исследование java.util. Этот важный пакет содержит большой набор классов и интерфейсов, которые поддерживают широкий спектр функциональных возможностей.

Например, в java.util есть классы, которые генерируют псевдослучайные числа, управляют датой и временем, поддерживают события, манипулируют наборами битов, разбивают строки на лексемы и обрабатывают форматированные данные. Пакет java.util также содержит одну из самых мощных подсистем Java: инфраструктуру коллекций *Collections Framework*, которая представляет собой сложную иерархию интерфейсов и классов, обеспечивающую современную технологию управления группами объектов. Она заслуживает пристального внимания всех программистов. Начиная с версии JDK 9, пакет java.util является частью модуля java.base.

Поскольку пакет java.util содержит широкий набор функций, он довольно большой. Ниже приведен список его классов верхнего уровня:

AbstractCollection	Formatter	PropertyPermission
AbstractList	GregorianCalendar	PropertyResourceBundle
AbstractMap	HashMap	Random
AbstractQueue	HashSet	ResourceBundle
AbstractSequentialList	Hashtable	Scanner
AbstractSet	HexFormat	ServiceLoader
ArrayDeque	IdentityHashMap	SimpleTimeZone
ArrayList	IntSummaryStatistics	Spliterators
Arrays	LinkedHashMap	SplitableRandom
Base64	LinkedHashSet	Stack
BitSet	LinkedList	StringJoiner
Calendar	ListResourceBundle	StringTokenizer
Collections	Locale	Timer
Currency	LongSummaryStatistics	TimerTask
Date	Objects	TimeZone

Dictionary	Observable (объявлен нерекомендуемым в JDK 9)	TreeMap
DoubleSummaryStatistics	Optional	TreeSet
EnumMap	OptionalDouble	UUID
EnumSet	OptionalInt	Vector
EventListenerProxy	OptionalLong	WeakHashMap
EventObject	PriorityQueue	
FormattableFlags	Properties	

Далее показаны интерфейсы, определенные в `java.util`:

Collection	Map.Entry	ServiceLoader.Provider
Comparator	NavigableMap	Set
Deque	NavigableSet	SortedMap
Enumeration	Observer (объявлен нерекомендуемым в JDK 9)	SortedSet
EventListener	PrimitiveIterator	Spliterator
Formattable	PrimitiveIterator.OfDouble	Spliterator.OfDouble
Iterator	PrimitiveIterator.OfInt	Spliterator.OfInt
List	PrimitiveIterator.OfLong	Spliterator.OfLong
ListIterator	Queue	Spliterator.OfPrimitive
Map	RandomAccess	

Из-за большого размера описание пакета `java.util` разбито на две главы. В данной главе рассматриваются члены `java.util`, которые являются частью Collections Framework. В главе 21 обсуждаются другие классы и интерфейсы `java.util`.

Обзор Collections Framework

Инфраструктура Collections Framework стандартизирует способ обработки групп объектов в программах. Коллекции не входили в исходный выпуск Java, а были добавлены в J2SE 1.2. До появления Collections Framework в Java для хранения групп объектов и управления ими предлагались специальные классы, такие как `Dictionary`, `Vector`, `Stack` и `Properties`. Хотя классы подобного рода были весьма полезными, им не хватало центральной объединяющей основы. Скажем, способ использования `Vector` отличался от способа работы с `Properties`. Кроме того, этот ранний специальный подход не проектировался так, чтобы быть легко расширяемым или адаптируемым. Ответом на упомянутые и на другие проблемы стала инфраструктура Collections Framework.

Инфраструктура Collections Framework была разработана для достижения нескольких целей. Во-первых, она должна была быть высокопроизводительной. Реализации фундаментальных коллекций (динамических массивов, связ-

ных списков, деревьев и хеш-таблиц) крайне эффективны. Вам редко (если вообще когда-либо) придется вручную писать код для одного из таких "механизмов данных". Во-вторых, инфраструктура должна была позволять различным типам коллекций работать одинаково и с высокой степенью взаимодействия. В-третьих, расширение и адаптация коллекций должна была быть простой. С этой целью вся инфраструктура Collections Framework построена на основе множества стандартных интерфейсов. Предлагается несколько стандартных реализаций таких интерфейсов (например, `LinkedList`, `HashSet` и `TreeSet`), которые разрешено применять в готовом виде. При желании можно реализовывать собственную коллекцию. Ради удобства созданы разнообразные реализации специального назначения, а также предоставляются частичные реализации, облегчающие создание собственного класса коллекции. Наконец, были добавлены механизмы, которые позволяют интегрировать стандартные массивы в Collections Framework.

Алгоритмы — еще одна важная часть механизма коллекций. Алгоритмы работают с коллекциями и определяются как статические методы в классе `Collections`. Таким образом, они доступны всем коллекциям. Каждый класс коллекции не обязательно должен реализовывать собственные версии этих методов. Алгоритмы обеспечивают стандартные средства управления коллекциями.

С инфраструктурой Collections Framework тесно связан еще один элемент — интерфейс `Iterator`. *Итератор* предлагает универсальный стандартизированный способ доступа к элементам внутри коллекции по одному за раз. Таким образом, итератор предоставляет средства *перечисления содержимого коллекции*. Поскольку каждая коллекция поддерживает итератор, доступ к элементам любого класса коллекции можно получить с помощью методов, определенных в интерфейсе `Iterator`. В итоге код, который циклически проходит по набору, после внесения небольших изменений также может использоваться, скажем, для циклического прохода по списку.

В версии JDK 8 добавлен еще один тип итератора, называемый *сплитератором* (spliterator), или разделяющим итератором. Выражаясь кратко, сплитераторы представляют собой итераторы, которые обеспечивают поддержку параллельной итерации. Сплитераторы поддерживаются интерфейсом `Spliterator` и рядом вложенных интерфейсов, поддерживающих примитивные типы. Также доступны интерфейсы итераторов, предназначенные для применения с примитивными типами, вроде `PrimitiveIterator` и `PrimitiveIterator.OfDouble`.

Помимо коллекций в Collections Framework определено несколько интерфейсов карт и классов. Карты хранят пары "ключ-значение". Невзирая на то что карты входят в состав Collections Framework, они не являются "коллекциями" в строгом смысле этого понятия. Однако для карты несложно получить *представление в виде коллекции*, которое содержит элементы карты, сохраненные в коллекции. Таким образом, при желании содержимое карты можно обрабатывать как коллекцию.

Механизм коллекций был модифицирован для некоторых исходных классов, определенных в `java.util`, чтобы их тоже можно было интегрировать в новую систему. Важно понимать, что хотя добавление коллекций изменило архитектуру многих исходных служебных классов, ни один из них не стал нерекомендуемым. Коллекции просто обеспечивают более эффективный способ решения ряда задач.

На заметку! Если вы знакомы с языком C++, то вам будет полезно узнать, что технология коллекций Java похожа по духу на библиотеку стандартных шаблонов (Standard Template Library — STL), определенную в C++. То, что в C++ называется контейнером, в Java называется коллекцией. Тем не менее, между Collections Framework и STL существуют значительные отличия. Важно не делать поспешных выводов.

Интерфейсы коллекций

В инфраструктуре Collections Framework определено несколько основных интерфейсов. Далее представлен обзор каждого интерфейса. Начинать с интерфейсов коллекций необходимо из-за того, что они определяют фундаментальную природу классов коллекций. Иными словами, конкретные классы просто предоставляют разные реализации стандартных интерфейсов. Интерфейсы, лежащие в основе коллекций, приведены в табл. 20.1.

Таблица 20.1. Интерфейсы, поддерживающие коллекции

Интерфейс	Описание
`Collection`	Позволяет работать с группами объектов; находится на вершине иерархии коллекций
`Deque`	Расширяет `Queue` для поддержки двусторонней очереди
`List`	Расширяет `Collection` для поддержки последовательностей (списков объектов)
`NavigableSet`	Расширяет `SortedSet` для поддержки извлечения элементов на базе поиска с наиболее точным совпадением
`Queue`	Расширяет `Collection` для поддержки специальных типов списков, в которых элементы удаляются только из головы
`Set`	Расширяет `Collection` для поддержки наборов, которые должны содержать уникальные элементы
`SortedSet`	Расширяет `Set` для поддержки сортированных наборов

Помимо интерфейсов коллекций в коллекциях также используются интерфейсы `Comparator`, `RandomAccess`, `Iterator`, `ListIterator` и `Spliterator`, которые подробно описаны далее в главе. Интерфейс `Comparator` опреде-

ляет, каким образом сравниваются два объекта; Iterator, ListIterator и Spliterator выполняют перечисление объектов внутри коллекции. Реализуя интерфейс RandomAccess, список указывает на то, что он поддерживает эффективный произвольный доступ к своим элементам.

Для обеспечения наибольшей гибкости в применении интерфейсы коллекций позволяют делать некоторые методы необязательными. Необязательные методы позволяют изменять содержимое коллекции. Коллекции, которые поддерживают эти методы, называются *модифицируемыми*. Коллекции, не допускающие изменения своего содержимого, называются *немодифицируемыми*. Если предпринимается попытка использовать один из этих методов в немодифицируемой коллекции, тогда генерируется исключение UnsupportedOperationException. Все встроенные коллекции являются модифицируемыми.

Интерфейсы коллекций рассматриваются в последующих разделах.

Интерфейс Collection

Интерфейс Collection — это основа, на которой построена инфраструктура Collections Framework, т.к. он должен быть реализован любым классом, определяющим коллекцию. Collection представляет собой обобщенный интерфейс с таким объявлением:

```
interface Collection<E>
```

С помощью E указывается тип объектов, которые будут храниться в коллекции. Интерфейс Collection расширяет Iterable, т.е. по всем коллекциям можно проходить в цикле for стиля "for-each". (Вспомните, что только классы, реализующие Iterable, допускают проход в цикле посредством for.)

В интерфейсе Collection объявлены основные методы, которые будут иметь все коллекции; они кратко описаны в табл. 20.2. Поскольку все коллекции реализуют интерфейс Collection, для четкого понимания инфраструктуры необходимо ознакомиться с его методами. Некоторые методы могут генерировать исключение UnsupportedOperationException. Как уже объяснялось, так происходит, если коллекцию изменять нельзя.

Исключение ClassCastException генерируется, когда один объект несовместим с другим, скажем, при попытке добавления в коллекцию несовместимого объекта. Исключение NullPointerException возникает, если предпринимается попытка сохранить объект null, а элементы null в коллекции не разрешены. Исключение IllegalArgumentException генерируется в случае применения недопустимого аргумента. Исключение IllegalStateException происходит при попытке добавления элемента в заполненную коллекцию фиксированной длины.

Таблица 20.2. Методы, объявленные в интерфейсе `Collection`

Метод	Описание
`boolean add(E obj)`	Добавляет `obj` в вызывающую коллекцию. Возвращает `true` в случае успешного добавления `obj`. Возвращает `false`, если `obj` уже является членом коллекции, а дубликаты в коллекции не разрешены
`boolean addAll (Collection <? extends E> c)`	Добавляет все элементы коллекции `c` в вызывающую коллекцию. Возвращает `true`, если коллекция изменилась (т.е. элементы были добавлены), или `false` в противном случае
`void clear()`	Удаляет из вызывающей коллекции все элементы
`boolean contains (Object obj)`	Возвращает `true`, если `obj` является элементом вызывающей коллекции, или `false` в противном случае
`boolean containsAll (Collection<?> c)`	Возвращает `true`, если вызывающая коллекция содержит все элементы коллекции `c`, или `false` в противном случае
`boolean equals (Object obj)`	Возвращает `true`, если вызывающая коллекция и `obj` равны. В противном случае возвращает `false`
`int hashCode()`	Возвращает хеш-код для вызывающей коллекции
`boolean isEmpty()`	Возвращает `true`, если вызывающая коллекция пуста, или `false` в противном случае
`Iterator<E> iterator()`	Возвращает итератор для вызывающей коллекции
`default Stream<E> parallelStream()`	Возвращает поток данных, который использует вызывающую коллекцию в качестве своего источника элементов. Если возможно, то поток данных поддерживает параллельные операции
`boolean remove (Object obj)`	Удаляет из вызывающей коллекции один экземпляр `obj`. Возвращает `true`, если элемент был удален, или `false` в противном случае
`boolean removeAll (Collection<?> c)`	Удаляет из вызывающей коллекции все элементы коллекции `c`. Возвращает `true`, если коллекция изменилась (т.е. элементы были удалены), или `false` в противном случае

Окончание табл. 20.2

Метод	Описание
`default boolean removeIf(Predicate <? super E> predicate)`	Удаляет из вызывающей коллекции элементы, которые удовлетворяют условию, указанному в `predicate`
`boolean retainAll (Collection<?> c)`	Удаляет из вызывающей коллекции все элементы кроме тех, что находятся в коллекции `c`. Возвращает `true`, если коллекция изменилась (т.е. элементы были удалены), или `false` в противном случае
`int size()`	Возвращает количество элементов, содержащихся в вызывающей коллекции
`default Spliterator<E> spliterator()`	Возвращает сплитератор для вызывающей коллекции
`default Stream<E> stream()`	Возвращает поток данных, который использует вызывающую коллекцию в качестве своего источника элементов. Поток данных является последовательным
`default <T> T[] toArray (IntFunction<T[]> arrayGen)`	Возвращает массив элементов из вызывающей коллекции. Возвращаемый массив создается функцией, указанной в `arrayGen`. Если любой элемент коллекции имеет тип, который не совместим с типом массива, тогда генерируется исключение `ArrayStoreException`
`Object[] toArray()`	Возвращает массив элементов из вызывающей коллекции
`<T> T[] toArray(T[] array)`	Возвращает массив элементов из вызывающей коллекции. Если размер массива равен количеству элементов в коллекции, то эти элементы возвращаются в массиве. Если размер массива меньше количества элементов в коллекции, тогда выделяется память под новый массив нужного размера, который затем возвращается. Если размер массива больше количества элементов в коллекции, тогда элементы массива, следующие после последнего элемента коллекции, устанавливаются в `null`. Если любой элемент коллекции имеет тип, который не совместим с типом массива, тогда генерируется исключение `ArrayStoreException`

Объекты добавляются в коллекцию с помощью вызова add(). Обратите внимание, что метод add() принимает аргумент типа E, а это значит, что объекты, добавляемые в коллекцию, должны быть совместимыми с типом данных, который ожидает коллекция. Вызвав метод addAll(), можно добавить все содержимое одной коллекции в другую.

Объект удаляется с использованием метода remove(). Для удаления группы объектов понадобится вызвать метод removeAll(). Чтобы удалить все элементы кроме входящих в указанную группу необходимо вызвать метод retainAll(). Для удаления элемента в случае, если он удовлетворяет некоторому условию, применяется метод removeIf(). Чтобы очистить коллекцию, нужно вызвать метод clear(). Для выяснения, содержит ли коллекция конкретный объект, используется метод contains(). Чтобы определить, содержит ли одна коллекция все члены другой, понадобится вызвать метод containsAll(). Для определения, пуста ли коллекция, предназначен метод isEmpty(). Количество элементов, находящихся в текущий момент внутри коллекции, позволяет выяснить метод size().

Методы toArray() возвращают массив, который содержит элементы, хранящиеся в коллекции. Первая форма toArray() возвращает массив типа Object. Вторая форма возвращает массив элементов того же типа, что и массив, указанный в качестве параметра. Обычно вторая форма более удобна, т.к. возвращает желаемый тип массива. В версии JDK 11 была добавлена третья форма toArray(), которая позволяет указать функцию, получающую массив. Эти методы важнее, чем может показаться на первый взгляд. Часто обработка содержимого коллекции с применением синтаксиса, подобного массивам, оказывается выгодной. Обеспечивая переходы между коллекциями и массивами, можно получить лучшее из обоих миров.

Две коллекции сравниваются на предмет равенства с помощью метода equals(). Точный смысл понятия "равенство" может отличаться от коллекции к коллекции. Например, метод equals() может быть реализован так, чтобы сравнивать значения элементов, хранящихся в коллекции, или же сравнивать ссылки на элементы коллекции.

Другим важным методом является iterator(), который возвращает итератор для коллекции. Метод spliterator() возвращает сплитератор для коллекции. Итераторы часто используются при работе с коллекциями. Наконец, методы stream() и parallelStream() возвращают поток, который задействует коллекцию как источник элементов. (Интерфейс Stream более подробно обсуждается в главе 30.)

Интерфейс List

Интерфейс List расширяет Collection и объявляет поведение коллекции, в которой хранится последовательность элементов. Элементы можно вставлять или получать к ним доступ по позиции в списке, применяя индекс, который начинается с нуля. Список может содержать повторяющиеся элементы. List является обобщенным интерфейсом со следующим объявлением:

```
interface List<E>
```

В E указывается тип объектов, которые будет хранить список.

Вдобавок к методам, определенным в Collection, в интерфейсе List определены собственные методы, кратко описанные в табл. 20.3. Еще раз обратите внимание, что некоторые из этих методов будут генерировать исключение UnsupportedOperationException, если список не может быть модифицирован, и ClassCastException, когда один объект оказывается несовместимым с другим, например, при попытке добавления в список несовместимого объекта. Кроме того, определенные методы генерируют исключение IndexOutOfBoundsException в случае использования недопустимого индекса. Исключение NullPointerException возникает, если предпринимается попытка сохранить объект null, а наличие в списке элементов null не разрешено. Исключение IllegalArgumentException генерируется при использовании недопустимого аргумента.

Таблица 20.3. Методы, объявленные в интерфейсе List

Метод	Описание
`void add(int index, E obj)`	Вставляет obj в вызывающий список по индексу, переданному в index. Ранее существовавшие элементы в точке вставки или после нее сдвигаются вправо. В итоге элементы списка не перезаписываются
`boolean addAll (int index, Collection <? extends E> c)`	Вставляет все элементы из коллекции c в вызывающий список по индексу, переданному в index. Ранее существовавшие элементы в точке вставки или после нее сдвигаются вправо. В итоге элементы списка не перезаписываются. Возвращает true, если вызывающий список изменился, или false в противном случае
`static <E> List<E> copyOf(Collection <? extends E> from)`	Возвращает список, который содержит те же элементы, которые указаны в коллекции from. Возвращаемый список является немодифицируемым и основанным на значении. Значения null не разрешены
`E get(int index)`	Возвращает объект, который хранится по указанному в index индексу внутри вызывающего списка
`int indexOf (Object obj)`	Возвращает индекс первого экземпляра obj в вызывающем списке. Если obj в списке отсутствует, тогда возвращается -1
`int lastIndexOf (Object obj)`	Возвращает индекс последнего экземпляра obj в вызывающем списке. Если obj в списке отсутствует, тогда возвращается -1

Окончание табл. 20.3

Метод	Описание
`ListIterator<E> listIterator()`	Возвращает итератор для вызывающего списка, который стартует с начала списка
`ListIterator<E> listIterator (int index)`	Возвращает итератор для вызывающего списка, который стартует с индекса, указанного в index
`static <E> List<E> of(`*список-параметров*`)`	Создает немодифицируемый список на основе значений, который содержит элементы, указанные в списке параметров. Значения null не разрешены. Предусмотрены многочисленные перегруженные версии. Детали будут обсуждаться далее
`E remove(int index)`	Удаляет из вызывающего списка элемент в позиции index и возвращает удаленный элемент. Результирующий список уплотняется, т.е. индексы последующих элементов уменьшаются на единицу
`default void replaceAll (UnaryOperator<E> opToApply)`	Обновляет каждый элемент в списке значением, полученным из функции opToApply
`E set(int index, E obj)`	Присваивает obj элементу по индексу, указанному в index внутри вызывающего списка. Возвращает старое значение
`default void sort (Comparator <? super E> comp)`	Сортирует список с использованием компаратора, указанного в comp
`List<E> subList (int start, int end)`	Возвращает список, который включает элементы от start до end-1 в вызывающем списке. На элементы в возвращаемом списке также ссылается и вызывающий объект

К версиям `add()` и `addAll()`, которые определены в `Collection`, интерфейс `List` добавляет методы `add(int, E)` и `addAll(int, Collection)`. Эти методы вставляют элементы по указанному индексу. Кроме того, интерфейс `List` изменяет семантику методов `add(E)` и `addAll(Collection)`, определенных в `Collection`, так что они добавляют элементы в конец списка. С помощью метода `replaceAll()` можно модифицировать все элементы коллекции.

Для получения объекта, хранящегося в определенной позиции, необходимо вызвать метод `get()` с индексом объекта. Чтобы присвоить значение

элементу в списке, нужно вызвать метод set() с указанием индекса объекта, подлежащего изменению. Для нахождения индекса объекта предназначены методы indexOf() и lastIndexOf().

Получить подсписок списка можно посредством метода subList(), передав ему начальный и конечный индексы подсписка. Как вы понимаете, метод subList() повышает удобство обработки списков. Один из способов сортировки списка предусматривает использование метода sort(), определенного в List.

Начиная с версии JDK 9, в интерфейсе List определен фабричный метод of(), который имеет несколько перегруженных версий. Каждая версия возвращает немодифицируемую коллекцию на основе значений, которая состоит из переданных ей аргументов. Основная цель of() заключается в том, чтобы предоставить удобный и эффективный способ создания небольшой коллекции типа List. Существуют 12 перегруженных версий of(), из которых одна не принимает аргументов и создает пустой список:

```
static <E> List<E> of()
```

Десять перегруженных версий принимают от одного до десяти аргументов и создают список, который содержит указанные элементы:

```
static <E> List<E> of(E obj1)
static <E> List<E> of(E obj1, E obj2)
static <E> List<E> of(E obj, E obj2, E obj3)
...
static <E> List<E> of(E ob1, E obj2, E obj3, E obj4, E obj5,
                      E obj6, E obj7, E obj8, E obj9, E obj10)
```

В последнем методе of() указываются аргументы переменной длины, позволяющие принимать произвольное количество элементов или массив элементов:

```
static <E> List<E> of(E ... objs)
```

Для всех версий элементы null не разрешены. Во всех случаях реализация List не указывается.

Интерфейс Set

Интерфейс Set определяет набор. Он расширяет интерфейс Collection и задает поведение коллекции, которое не допускает дублирования элементов, поэтому метод add() возвращает false, если предпринимается попытка добавления в набор повторяющихся элементов. За двумя исключениями собственные дополнительные методы в Set не определены. Set представляет собой обобщенный интерфейс со следующим объявлением:

```
interface Set<E>
```

В E указывается тип объектов, которые будут храниться внутри набора.

Начиная с версии JDK 9, интерфейс Set содержит фабричный метод of(), который имеет перегруженных версий. Каждая версия возвращает немоди-

фицируемую коллекцию на основе значений, которая состоит из переданных ей аргументов. Основная цель `of()` заключается в том, чтобы предоставить удобный и эффективный способ создания небольшой коллекции типа `Set`. Существует 12 перегруженных версий `of()`, из которых одна не принимает аргументов и создает пустой набор:

```
static <E> Set<E> of()
```

Десять перегруженных версий принимают от одного до десяти аргументов и создают набор, который содержит указанные элементы:

```
static <E> Set<E> of(E obj1)
static <E> Set<E> of(E obj1, E obj2)
static <E> Set<E> of(E obj, E obj2, E obj3)
...
static <E> Set<E> of(E ob1, E obj2, E obj3, E obj4, E obj5,
                     E obj6, E obj7, E obj8, E obj9, E obj10)
```

В последнем методе `of()` указываются аргументы переменной длины, позволяющие принимать произвольное количество элементов или массив элементов:

```
static <E> Set<E> of(E ... objs)
```

Для всех версий элементы `null` не разрешены. Во всех случаях реализация `Set` не указывается.

Начиная с версии JDK 10, интерфейс `Set` включает статический метод `copyOf()`:

```
static <E> Set<E> copyOf(Collection <? extends E> from)
```

Он возвращает набор, который содержит те же элементы, что и в коллекции `from`. Значения `null` не разрешены. Возвращаемый набор является немодифицируемым и основанным на значении.

Интерфейс SortedSet

Интерфейс `SortedSet` расширяет `Set` и обеспечивает поведение набора, отсортированного в возрастающем порядке. `SortedSet` — обобщенный интерфейс со следующим объявлением:

```
interface SortedSet<E>
```

В `E` указывается тип объектов, которые будут храниться внутри набора.

В дополнение к методам, предоставляемым `Set`, в интерфейсе `SortedSet` объявлены методы, кратко описанные в табл. 20.4. Некоторые методы генерируют исключение `NoSuchElementException`, если вызывающий набор не содержит элементов. Исключение `ClassCastException` возникает, когда объект несовместим с элементами набора. Исключение `NullPointerException` генерируется при попытке использования объекта `null`, наличие которого внутри набора не разрешено. Исключение `IllegalArgumentException` возникает, если применяется недопустимый аргумент.

Таблица 20.4. Методы, объявленные в интерфейсе SortedSet

Метод	Описание
`Comparator <? super E> comparator()`	Возвращает компаратор вызывающего отсортированного набора. Если для этого набора используется естественное упорядочение, тогда возвращается `null`
`E first()`	Возвращает первый элемент в вызывающем отсортированном наборе
`SortedSet<E> headSet(E end)`	Возвращает объект `SortedSet` с элементами, предшествующими `end`, которые содержатся в вызывающем отсортированном наборе. На элементы в возвращаемом отсортированном наборе также ссылается и вызывающий отсортированный набор
`E last()`	Возвращает последний элемент в вызывающем отсортированном наборе
`SortedSet<E> subSet(E start, E end)`	Возвращает объект `SortedSet`, который содержит элементы между `start` и `end-1`. На элементы в возвращаемом отсортированном наборе также ссылается и вызывающий отсортированный набор
`SortedSet<E> tailSet(E start)`	Возвращает объект `SortedSet` с элементами, начиная с позиции `start` и до конца, которые содержатся в вызывающем отсортированном наборе. На элементы в возвращаемом отсортированном наборе также ссылается и вызывающий отсортированный набор

В интерфейсе `SortedSet` определено несколько методов, повышающих удобство обработки наборов. Получить первый объект в наборе позволяет метод `first()`, а последний элемент — метод `last()`. Для получения поднабора отсортированного набора понадобится вызвать метод `subSet()`, указав первый и последний объекты внутри набора. Если требуется поднабор, который начинается с первого элемента набора, тогда необходимо использовать метод `headSet()`, а если интересует поднабор, простирающийся до конца набора, то следует применять метод `tailSet()`.

Интерфейс NavigableSet

Интерфейс `NavigableSet` расширяет `SortedSet` и обеспечивает поведение коллекции, которая поддерживает извлечение элементов на основе наиболее точного совпадения с заданным значением или значениями. `NavigableSet` — обобщенный интерфейс с показанным ниже объявлением:

```
interface NavigableSet<E>
```

В `E` указывается тип объектов, которые будут храниться внутри набора. Вдобавок к методам, унаследованным от `SortedSet`, в интерфейсе `NavigableSet` объявлены методы, кратко описанные в табл. 20.5.

Таблица 20.5. Методы, объявленные в интерфейсе NavigableSet

Метод	Описание
`E ceiling(E obj)`	Ищет внутри набора наименьший элемент e, который больше или равен obj. Если такой элемент найден, то он возвращается. В противном случае возвращается null
`Iterator<E> descendingIterator()`	Возвращает итератор, который перемещается от наибольшего элемента к наименьшему. Другими словами, возвращается обратный итератор
`NavigableSet<E> descendingSet()`	Возвращает объект NavigableSet, противоположный вызывающему набору. Результирующий набор поддерживается в актуальном состоянии вызывающим набором
`E floor(E obj)`	Ищет внутри набора наибольший элемент e, который меньше или равен obj. Если такой элемент найден, то он возвращается. В противном случае возвращается null
`NavigableSet<E> headSet(E upperBound, boolean incl)`	Возвращает объект NavigableSet, содержащий все элементы из вызывающего набора, которые меньше upperBound. Если incl равно true, тогда включается элемент, равный upperBound. Результирующий набор поддерживается в актуальном состоянии вызывающим набором
`E higher(E obj)`	Ищет внутри набора наименьший элемент e, который больше obj. Если такой элемент найден, то он возвращается. В противном случае возвращается null
`E lower(E obj)`	Ищет внутри набора наибольший элемент e, который меньше obj. Если такой элемент найден, то он возвращается. В противном случае возвращается null
`E pollFirst()`	Возвращает первый элемент, в процессе удаляя его. Поскольку набор отсортирован, это будет элемент с наименьшим значением. В случае пустого набора возвращается null
`E pollLast()`	Возвращает последний элемент, в процессе удаляя его. Поскольку набор отсортирован, это будет элемент с наибольшим значением. В случае пустого набора возвращается null

Окончание табл. 20.5

Метод	Описание
`NavigableSet<E> subSet(E lowerBound, boolean lowIncl, E upperBound, boolean highIncl)`	Возвращает объект `NavigableSet`, содержащий все элементы из вызывающего набора, которые больше `lowerBound` и меньше `upperBound`. Если `lowIncl` равно `true`, тогда включается элемент, равный `lowerBound`. Если `highIncl` равно `true`, тогда включается элемент, равный `upperBound`. Результирующий набор поддерживается в актуальном состоянии вызывающим набором
`NavigableSet<E> tailSet(E lowerBound, boolean incl)`	Возвращает объект `NavigableSet`, содержащий все элементы из вызывающего набора, которые больше `lowerBound`. Если `incl` равно `true`, тогда включается элемент, равный `lowerBound`. Результирующий набор поддерживается в актуальном состоянии вызывающим набором

Если объект несовместим с элементами внутри набора, тогда генерируется исключение `ClassCastException`. При попытке использования объекта `null`, когда такие объекты не разрешены, возникает исключение `NullPointerException`. В случае применения недопустимого аргумента генерируется исключение `IllegalArgumentException`.

Интерфейс Queue

Интерфейс `Queue` расширяет `Collection` и обеспечивает поведение очереди, которая часто представляет собой список, действующий по принципу "первым пришел — первым обслужен". Однако существуют виды очередей, в которых упорядочение основано на других критериях. `Queue` — обобщенный интерфейс с таким объявлением:

```
interface Queue<E>
```

В `E` указывается тип объектов, которые будут храниться в очереди. В табл. 20.6 кратко описаны методы, объявленные в интерфейсе `Queue`.

Некоторые методы генерируют `ClassCastException`, когда объект несовместим с элементами в очереди. Исключение `NullPointerException` возникает, если предпринимается попытка сохранить объект `null`, а элементы `null` в очереди не разрешены. Исключение `IllegalArgumentException` происходит в случае использования недопустимого аргумента.

Исключение `IllegalStateException` генерируется при попытке добавления элемента в заполненную очередь фиксированной длины. Исключение `NoSuchElementException` возникает при попытке удаления элемента из пустой очереди.

Таблица 20.6. Методы, объявленные в интерфейсе Queue

Метод	Описание
`E element()`	Возвращает элемент из головы очереди, не удаляя его. Если очередь пуста, тогда генерируется исключение `NoSuchElementException`
`boolean offer(E obj)`	Предпринимает попытку добавления `obj` в очередь. Возвращает `true`, если `obj` был добавлен, или `false` в противном случае
`E peek()`	Возвращает элемент из головы очереди, не удаляя его. Если очередь пуста, тогда возвращается `null`
`E poll()`	Возвращает элемент из головы очереди, в процессе удаляя его. Если очередь пуста, тогда возвращается `null`
`E remove()`	Удаляет элемент из головы очереди, возвращая его. Если очередь пуста, тогда генерируется исключение `NoSuchElementException`

Несмотря на свою простоту, интерфейс `Queue` интересен несколькими аспектами. Во-первых, элементы можно удалять только из головы очереди. Во-вторых, существуют два метода для получения и удаления элементов: `poll()` и `remove()`. Разница между ними в том, что `poll()` возвращает `null`, если очередь пуста, а `remove()` генерирует исключение. В-третьих, есть два метода, `element()` и `peek()`, которые получают элемент в голове очереди, но не удаляют его. Они отличаются лишь тем, что в случае пустой очереди метод `element()` генерирует исключение, а `peek()` возвращает `null`. Наконец, обратите внимание, что метод `offer()` только пытается добавить элемент в очередь. Поскольку некоторые очереди имеют фиксированную длину и могут быть заполнены, метод `offer()` может завершиться неудачей.

Интерфейс Deque

Интерфейс `Deque` расширяет `Queue` и обеспечивает поведение двусторонней очереди. Двусторонние очереди могут функционировать как стандартные очереди "первым пришел — первым обслужен" или как стеки "последний пришел — первым обслужен". `Deque` — обобщенный интерфейс со следующим объявлением:

```
interface Deque<E>
```

В `E` указывается тип объектов, которые будут храниться в двусторонней очереди. Вдобавок к методам, унаследованным от `Queue`, в интерфейсе `Deque` объявлены методы, кратко описанные в табл. 20.7. Некоторые методы генерируют исключение `ClassCastException`, когда объект несовместим с элементами в очереди. Исключение `NullPointerException` возникает, если предпринимается попытка сохранить объект `null`, а элементы `null` в дву-

сторонней очереди не разрешены. Исключение `IllegalArgumentException` генерируется в случае применения недопустимого аргумента. Исключение `IllegalStateException` возникает при попытке добавления элемента в двустороннюю заполненную очередь фиксированной длины. Исключение `NoSuchElementException` генерируется при попытке удаления элемента из пустой очереди.

Таблица 20.7. Методы, объявленные в интерфейсе Deque

Метод	Описание
`void addFirst(E obj)`	Добавляет `obj` в голову двусторонней очереди. Генерирует исключение `IllegalStateException`, если в двусторонней очереди с ограниченной емкостью отсутствует свободное пространство
`void addLast(E obj)`	Добавляет `obj` в хвост двусторонней очереди. Генерирует исключение `IllegalStateException`, если в двусторонней очереди с ограниченной емкостью отсутствует свободное пространство
`Iterator<E> descendingIterator()`	Возвращает итератор, который перемещается от хвоста к голове двусторонней очереди. Другими словами, возвращается обратный итератор
`E getFirst()`	Возвращает первый элемент из двусторонней очереди, не удаляя его. Если двусторонняя очередь пуста, тогда генерируется исключение `NoSuchElementException`
`E getLast()`	Возвращает последний элемент из двусторонней очереди, не удаляя его. Если двусторонняя очередь пуста, тогда генерируется исключение `NoSuchElementException`
`boolean offerFirst (E obj)`	Предпринимает попытку добавления `obj` в голову двусторонней очереди. Возвращает `true`, если `obj` был добавлен, или `false` в противном случае. Таким образом, этот метод возвращает `false` при попытке добавить `obj` в переполненную очередь с ограниченной емкостью
`boolean offerLast (E obj)`	Предпринимает попытку добавления `obj` в хвост двусторонней очереди. Возвращает `true`, если `obj` был добавлен, или `false` в противном случае

Окончание табл. 20.7

Метод	Описание
`E peekFirst()`	Возвращает элемент из головы двусторонней очереди, не удаляя его. Если двусторонняя очередь пуста, тогда возвращается `null`
`E peekLast()`	Возвращает элемент из хвоста двусторонней очереди, не удаляя его. Если двусторонняя очередь пуста, тогда возвращается `null`
`E pollFirst()`	Возвращает элемент из головы двусторонней очереди, в процессе удаляя его. Если двусторонняя очередь пуста, тогда возвращается `null`
`E pollLast()`	Возвращает элемент из хвоста двусторонней очереди, в процессе удаляя его. Если двусторонняя очередь пуста, тогда возвращается `null`
`E pop()`	Возвращает элемент из головы двусторонней очереди, в процессе удаляя его. Если двусторонняя очередь пуста, тогда генерируется исключение `NoSuchElementException`
`void push(E obj)`	Добавляет `obj` в голову двусторонней очереди. Генерирует исключение `IllegalStateException`, если в двусторонней очереди с ограниченной емкостью отсутствует свободное пространство
`E removeFirst()`	Возвращает элемент из головы двусторонней очереди, в процессе удаляя его. Если двусторонняя очередь пуста, тогда генерируется исключение `NoSuchElementException`
`boolean removeFirstOccurrence (Object obj)`	Удаляет первое вхождение `obj` из двусторонней очереди. Возвращает `true` в случае успеха и `false`, если двусторонняя очередь не содержит `obj`
`E removeLast()`	Возвращает элемент из хвоста двусторонней очереди, в процессе удаляя его. Если двусторонняя очередь пуста, тогда генерируется исключение `NoSuchElementException`
`boolean removeLastOccurrence (Object obj)`	Удаляет последнее вхождение `obj` из двусторонней очереди. Возвращает `true` в случае успеха и `false`, если двусторонняя очередь не содержит `obj`

Обратите внимание, что интерфейс Deque включает в себя методы push() и pop(), которые позволяют ему функционировать в качестве стека. Также взгляните на метод descendingIterator(), который возвращает итератор, выдающий элементы в обратном порядке. Другими словами, descendingIterator() возвращает итератор, который перемещается от конца коллекции к ее началу. Реализация интерфейса Deque может быть *ограниченной по емкости*, т.е. в двустороннюю очередь разрешается добавлять лимитированное количество элементов. В таком случае попытка добавления элемента в очередь может потерпеть неудачу. Интерфейс Deque позволяет обрабатывать такой отказ двумя способами. Во-первых, методы вроде addFirst() и addLast() генерируют исключение IllegalStateException, если двусторонняя очередь с ограниченной емкостью заполнена. Во-вторых, методы наподобие offerFirst() и offerLast() возвращают false, если элемент не может быть добавлен.

Классы коллекций

Теперь, когда вы знакомы с интерфейсами коллекций, можно приступать к изучению стандартных классов, которые их реализуют. Некоторые классы предоставляют полные реализации, которые можно использовать в том виде как есть. Другие являются абстрактными, обеспечивая базовые реализации, которые применяются в качестве отправных точек для создания конкретных коллекций. Обычно классы коллекций не синхронизируются, но позже в главе вы увидите, что можно получать синхронизированные версии.

Основные классы коллекций кратко описаны в табл. 20.8.

Таблица 20.8. Основные классы коллекций

Класс	Описание
AbstractCollection	Реализует большую часть интерфейса Collection
AbstractList	Расширяет AbstractCollection и реализует большую часть интерфейса List
AbstractQueue	Расширяет AbstractCollection и реализует часть интерфейса Queue
AbstractSequentialList	Расширяет AbstractList для использования коллекцией, которая применяет последовательный, а не произвольный доступ к своим элементам
LinkedList	Реализует связный список, расширяя AbstractSequentialList
ArrayList	Реализует динамический массив, расширяя AbstractList

Окончание табл. 20.8

Класс	Описание
ArrayDeque	Реализует динамическую двустороннюю очередь, расширяя AbstractCollection и реализуя интерфейс Deque
AbstractSet	Расширяет AbstractCollection и реализует большую часть интерфейса Set
EnumSet	Расширяет AbstractSet для использования с элементами перечислений
HashSet	Расширяет AbstractSet для применения с хеш-таблицей
LinkedHashSet	Расширяет HashSet, чтобы сделать возможной итерацию в порядке вставки
PriorityQueue	Расширяет AbstractQueue для поддержки очереди на основе приоритетов
TreeSet	Реализует набор, хранящийся в дереве. Расширяет AbstractSet

В последующих разделах рассматриваются конкретные классы коллекций и приводятся примеры их использования.

На заметку! Помимо классов, представляющих коллекции, для поддержки коллекций были перепроектированы такие унаследованные классы, как Vector, Stack и Hashtable. Все они рассматриваются далее в главе.

Класс ArrayList

Класс ArrayList является обобщенным классом, который расширяет AbstractList и реализует интерфейс List. Вот его объявление:

```
class ArrayList<E>
```

В E указывается тип объектов, которые будет хранить список.

Класс ArrayList поддерживает динамические массивы, которые по мере необходимости могут расти. Стандартные массивы в Java имеют фиксированную длину. После создания массивы не могут увеличиваться или уменьшаться, а потому необходимо заранее знать, сколько элементов будет содержать массив. Но иногда точный размер массива не известен вплоть до времени выполнения. Для ситуаций подобного рода в инфраструктуре Collections Framework предусмотрен класс ArrayList. По существу экземпляр ArrayList представляет собой массив объектных ссылок переменной длины, т.е. он способен динамически увеличиваться или уменьшаться в размерах.

Списковые массивы создаются с начальным размером. В случае превышения этого размера массив автоматически увеличивается, а когда объекты удаляются, размер массива может быть уменьшен.

На заметку! Динамические массивы также поддерживаются унаследованным классом `Vector`, который описан далее в главе.

В классе `ArrayList` определены следующие конструкторы:

```
ArrayList()
ArrayList(Collection<? extends E> c)
ArrayList(int capacity)
```

Первый конструктор строит пустой списковый массив. Второй конструктор создает списковый массивов, который инициализируется элементами коллекции `c`. Третий конструктор строит списковый массив с указанной в `capacity` начальной емкостью. Емкость — это размер лежащего в основе массива, который применяется для хранения элементов. По мере добавления элементов в списковый массив емкость автоматически увеличивается.

В показанной ниже программе демонстрируется простой случай использования `ArrayList`. Для объектов типа `String` создается списковый массив, в который затем добавляется несколько строк. (Вспомните, что строка в кавычках преобразуется в объект `String`.) Далее список отображается. После удаления ряда элементов список отображается снова.

```
// Демонстрация использования ArrayList.
import java.util.*;
class ArrayListDemo {
  public static void main(String[] args) {
    // Создать списковый массив.
    ArrayList<String> al = new ArrayList<String>();

    System.out.println("Начальный размер al: " + al.size());

    // Добавить элементы в списковый массив.
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");
    al.add(1, "A2");
    System.out.println("Размер al после добавления элементов: " + al.size());
    // Отобразить списковый массив.
    System.out.println("Содержимое al: " + al);

    // Удалить элементы из спискового массива.
    al.remove("F");
    al.remove(2);
    System.out.println("Размер al после удаления элементов: " + al.size());
    System.out.println("Содержимое al: " + al);
  }
}
```

Вот вывод, генерируемый программой:

```
Начальный размер: 0
Размер al после добавления элементов: 7
Содержимое al: [C, A2, A, E, B, D, F]
Размер al после удаления элементов: 5
Содержимое al: [C, A2, E, B, D]
```

Обратите внимание, что списковый массив `al` сначала пуст и растет по мере добавления в него элементов. При удалении элементов его размер уменьшается.

В предыдущем примере содержимое коллекции отображается с применением стандартного преобразования, которое обеспечивается методом `toString()`, унаследованным от `AbstractCollection`. Хотя для коротких примеров программ такой подход совершенно приемлем, метод `toString()` редко используется для отображения содержимого реальной коллекции. Обычно будут предоставляться собственные процедуры вывода. Но в приведенных ниже примерах вполне достаточно стандартного вывода, который создает `toString()`.

Несмотря на автоматическое увеличение емкости объекта `ArrayList` по ходе сохранения в нем объектов, увеличить емкость объекта `ArrayList` можно вручную, вызвав метод `ensureCapacity()`. Так поступают, когда заранее известно, что в списковом массиве `ArrayList` будет находиться гораздо больше элементов, чем он может вместить в текущий момент. За счет увеличения его емкости в самом начале удастся предотвратить несколько повторных выделений памяти позже. Поскольку повторные выделения памяти сопряжены с затратами в плане времени, предотвращение ненужных повторных выделений увеличивает производительность. Ниже приведена сигнатура метода `ensureCapacity()`:

```
void ensureCapacity(int cap)
```

В `cap` указывается новая минимальная емкость спискового массива.

И наоборот, если необходимо уменьшить размер массива, который лежит в основе объекта `ArrayList`, чтобы он был точно таким же, как количество содержащихся в нем элементов, понадобится вызвать метод `trimToSize()`:

```
void trimToSize()
```

Получение массива из `ArrayList`

При работе с классом `ArrayList` иногда нужно получить фактический массив, который хранит содержимое списка. Для этого можно вызвать метод `toArray()`, определенный в интерфейсе `Collection`. Существует несколько причин преобразования коллекции в массив, например:

- для ускорения обработки определенных операций;
- для передачи массива методу, который не перегружен для приема коллекции;
- для интеграции кода на основе коллекций с унаследованным кодом, который не воспринимает коллекции.

Какой бы ни была причина, преобразование ArrayList в массив — дело тривиальное.

Как объяснялось ранее, существуют три версии toArray(), которые ради напоминания показаны еще раз:

```
object[] toArray()
<T> T[] toArray(T[] array)
default <T> T[] toArray(IntFunction<T[]> arrayGen)
```

Первая форма конструктора возвращает массив элементов типа Object. Вторая и третья формы возвращают массив элементов типа T. По причине удобства далее будет применяться вторая форма, что и демонстрируется в следующей программе.

```
// Преобразование ArrayList в массив.
import java.util.*;

class ArrayListToArray {
  public static void main(String[] args) {
    // Создать списковый массив.
    ArrayList<Integer> al = new ArrayList<Integer>();

    // Добавить элементы в списковый массив.
    al.add(1);
    al.add(2);
    al.add(3);
    al.add(4);
    System.out.println("Содержимое массива al: " + al);

    // Получить массив.
    Integer[] ia = new Integer[al.size()];
    ia = al.toArray(ia);
    int sum = 0;

    // Просуммировать элементы массива.
    for(int i : ia) sum += i;
    System.out.println("Сумма элементов массива: " + sum);
  }
}
```

Вот вывод, генерируемый программой:

```
Содержимое массива al: [1, 2, 3, 4]
Сумма элементов массива: 10
```

Программа начинается с создания коллекции целых чисел. Затем вызывается метод toArray(), позволяющий получить массив целых чисел. Далее содержимое этого массива суммируется с помощью цикла for в стиле "for-each".

В примере программы есть еще кое-что интересное. Как вам известно, коллекции могут хранить только ссылки, а не значения примитивных типов. Тем не менее, автоупаковка позволяет передавать значения типа int в метод add() без необходимости помещать их внутрь объектов Integer вручную, что и показано в программе. Автоупаковка обеспечивает автоматическое помещение таких значений в надлежащие оболочки. Таким образом, автоупаковка значительно упрощает использование коллекций для хранения значений примитивных типов.

Класс LinkedList

Класс LinkedList расширяет AbstractSequentialList, а также реализует интерфейсы List, Deque и Queue. Он предоставляет структуру данных типа связного списка. LinkedList — обобщенный класс со следующим объявлением:

```
class LinkedList<E>
```

В E указывается тип объектов, которые будут храниться в связном списке. В классе LinkedList определены два конструктора:

```
LinkedList()
LinkedList(Collection<? extends E> c)
```

Первый конструктор создает пустой связный список. Второй конструктор строит связный список, который инициализируется элементами из коллекции c.

Поскольку класс LinkedList реализует интерфейс Deque, доступны методы, определенные в Deque. Для добавления элемента в начало списка применяются методы addFirst() и offerFirst(). Для добавления элементов в конец списка используются методы addLast() и offerLast(). Для получения первого элемента предназначены методы getFirst() и peekFirst(). Для получения последнего элемента применяются методы getLast() и peekLast(). Для удаления первого элемента используются методы removeFirst() и pollFirst(). Для удаления последнего элемента предназначены методы removeLast() и pollLast().

В показанной ниже программе иллюстрируется применение класса LinkedList:

```
// Демонстрация использования LinkedList.
import java.util.*;

class LinkedListDemo {
  public static void main(String[] args) {
    // Создать связный список.
    LinkedList<String> ll = new LinkedList<String>();

    // Добавить элементы в связный список.
    ll.add("F");
    ll.add("B");
    ll.add("D");
    ll.add("E");
    ll.add("C");
    ll.addLast("Z");
    ll.addFirst("A");
    ll.add(1, "A2");

    System.out.println("Исходное содержимое ll: " + ll);

    // Удалить элементы из связного списка.
    ll.remove("F");
    ll.remove(2);

    System.out.println("Содержимое ll после удаления: " + ll);
```

```
    // Удалить первый и последний элементы.
    ll.removeFirst();
    ll.removeLast();

    System.out.println("Содержимое ll после удаления первого и последнего
элементов: " + ll);

    // Получить и установить значение.
    String val = ll.get(2);
    ll.set(2, val + " изменено");

    System.out.println("Содержимое ll после изменения: " + ll);
  }
}
```

Вывод, генерируемый программой, выглядит следующим образом:

```
Исходное содержимое ll: [A, A2, F, B, D, E, C, Z]
Содержимое ll после удаления: [A, A2, D, E, C, Z]
Содержимое ll после удаления первого и последнего элементов: [A2, D, E, C]
Содержимое ll после изменения: [A2, D, E изменено, C]
```

Из-за того, что класс `LinkedList` реализует интерфейс `List`, вызовы `add(E)` добавляют элементы в конец списка подобно вызовам `addLast()`. Для вставки элементов в определенное место предназначена форма `add(int, E)` метода `add()`, как демонстрировал вызов `add(1, "A2")` в примере.

Обратите внимание, что третий элемент в `ll` изменяется с помощью вызовов `get()` и `set()`. Чтобы получить текущее значение элемента, методу `get()` необходимо передать индекс, в котором хранится элемент. Чтобы присвоить новое значение этому индексу, методу `set()` нужно передать индекс и само новое значение.

Класс HashSet

Класс `HashSet` расширяет `AbstractSet` и реализует интерфейс `Set`. Он создает коллекцию, в которой для хранения данных применяется хеш-таблица. `HashSet` — обобщенный класс с таким объявлением:

```
class HashSet<E>
```

В `E` указывается тип объектов, которые будет хранить набор.

Как, по всей видимости, известно большинству читателей, хеш-таблица хранит информацию с помощью механизма под названием *хеширование*. При хешировании информационное содержимое ключа используется для определения уникального значения, которое называется его *хеш-кодом*. Впоследствии хеш-код используется в качестве индекса, в котором хранятся данные, связанные с ключом. Преобразование ключа в его хеш-код выполняется автоматически — вам никогда не придется видеть сам хеш-код. Кроме того, напрямую обращаться по индексу в хеш-таблицу из кода нельзя. Преимущество хеширования заключается в том, что оно обеспечивает постоянство времени выполнения методов `add()`, `contains()`, `remove()` и `size()` даже для крупных хеш-таблиц.

Ниже перечислены конструкторы, которые определены в классе HashSet:

```
HashSet()
HashSet(Collection<? extends E> c)
HashSet(int capacity)
HashSet(int capacity, float fillRatio)
```

Первая форма создает стандартную хеш-таблицу. Вторая форма инициализирует хеш-таблицу с применением элементов из коллекции c. Третья форма инициализирует емкость хеш-таблицы значением, указанным в capacity. (Стандартная емкость равна 16.) Четвертая форма инициализирует как емкость, так и коэффициент заполнения (также называемый *коэффициентом загрузки*) хеш-таблицы на основе переданных аргументов. Значение коэффициента загрузки должно находиться между 0.0 и 1.0; оно определяет, насколько полной может быть хеш-таблица, прежде чем ее размер будет увеличен. В частности, когда количество элементов становится больше емкости хеш-таблицы, умноженной на коэффициент заполнения, хеш-таблица расширяется. Для конструкторов, не принимающих коэффициент заполнения, используется значение 0.75.

Никаких дополнительных методов кроме тех, которые предоставлены суперклассами и интерфейсами, в классе HashSet не определено.

Важно отметить, что класс HashSet не гарантирует порядок следования своих элементов, потому что процесс хеширования обычно не позволяет создавать отсортированные наборы. Если вам нужно отсортированное хранилище, тогда лучше выбрать другую коллекцию, скажем, TreeSet.

Вот пример, где демонстрируется применение HashSet:

```
// Демонстрация работы HashSet.
import java.util.*;

class HashSetDemo {
  public static void main(String[] args) {
    // Создать хеш-таблицу.
    HashSet<String> hs = new HashSet<String>();

    // Добавить элементы в хеш-таблицу.
    hs.add("Beta");
    hs.add("Alpha");
    hs.add("Eta");
    hs.add("Gamma");
    hs.add("Epsilon");
    hs.add("Omega");

    System.out.println(hs);
  }
}
```

Вывод, генерируемый программой, выглядит следующим образом:

```
[Gamma, Eta, Alpha, Epsilon, Omega, Beta]
```

Как уже объяснялось, элементы не хранятся в отсортированном порядке, так что точный вывод может варьироваться.

Класс LinkedHashSet

Класс LinkedHashSet расширяет HashSet, не добавляя собственные члены. Он является обобщенным классом с показанным далее объявлением:

```
class LinkedHashSet<E>
```

В E задается тип объектов, которые будет хранить набор. Конструкторы LinkedHashSet аналогичны конструкторам HashSet.

Класс LinkedHashSet поддерживает связный список элементов внутри набора в порядке их добавления, что делает возможной итерацию по набору в этом порядке. Другими словами, при циклическом переборе LinkedHashSet с использованием итератора элементы будут возвращаться в том порядке, в котором они были вставлены. Кроме того, в таком же порядке они содержатся в строке, возвращаемой методом toString() при его вызове на объекте LinkedHashSet. Чтобы увидеть эффект от LinkedHashSet, попробуйте в предыдущей программе заменить HashSet на LinkedHashSet. Результат будет отражать порядок вставки элементов:

```
[Beta, Alpha, Eta, Gamma, Epsilon, Omega]
```

Класс TreeSet

Класс TreeSet расширяет AbstractSet и реализует интерфейс NavigableSet. Он создает коллекцию, которая в качестве хранилища использует дерево. Объекты хранятся в отсортированном порядке по возрастанию. Доступ и извлечение производится довольно быстро, что делает класс TreeSet превосходным вариантом для организации хранения больших объемов отсортированной информации, в которой необходим быстрый поиск.

TreeSet — обобщенный класс со следующим объявлением:

```
class TreeSet<E>
```

В E указывается тип объектов, которые будут храниться внутри набора.

Ниже перечислены конструкторы TreeSet:

```
TreeSet()
TreeSet(Collection<? extends E> c)
TreeSet(Comparator<? super E> comp)
TreeSet(SortedSet<E> ss)
```

Первая форма конструктора строит пустой древовидный набор, который будет отсортирован в порядке возрастания в соответствии с естественным упорядочением его элементов. Вторая форма строит древовидный набор, содержащий элементы из коллекции c. Третья форма строит пустой древовидный набор, отсортированный в соответствии с компаратором, который указан в comp. (Компараторы рассматриваются далее в главе.) Четвертая форма строит древовидный набор, содержащий элементы из ss.

Рассмотрим пример применения класса TreeSet:

```
// Демонстрация использования TreeSet.
import java.util.*;

class TreeSetDemo {
```

```
  public static void main(String[] args) {
    // Создать древовидный набор.
    TreeSet<String> ts = new TreeSet<String>();
    // Добавить элементы в древовидный набор.
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");
    System.out.println(ts);
  }
}
```

Вот вывод, генерируемый программой:

```
[A, B, C, D, E, F]
```

Как упоминалось ранее, поскольку класс `TreeSet` хранит свои элементы в дереве, они автоматически располагаются в отсортированном порядке, что подтверждается выводом.

Так как `TreeSet` реализует интерфейс `NavigableSet`, для извлечения элементов `TreeSet` можно использовать методы, определенные в `NavigableSet`. Например, продолжим предыдущую программу. В следующем операторе с помощью метода `subSet()` получается поднабор `ts`, содержащий элементы между C (включительно) и F (исключительно), после чего результирующий набор отображается.

```
System.out.println(ts.subSet("C", "F"));
```

Вот вывод, генерируемый новым оператором:

```
[C, D, E]
```

Можете поэкспериментировать и с другими методами, которые определены в `NavigableSet`.

Класс `PriorityQueue`

Класс `PriorityQueue` расширяет `AbstractQueue` и реализует интерфейс `Queue`. Он создает очередь с приоритетом на основе компаратора очереди. `PriorityQueue` — обобщенный класс со следующим объявлением:

```
class PriorityQueue<E>
```

В `E` указывается тип объектов, хранящихся в очереди. Экземпляры `PriorityQueue` являются динамическими и растут по мере необходимости.

В классе `PriorityQueue` определены семь конструкторов:

```
PriorityQueue()
PriorityQueue(int capacity)
PriorityQueue(Comparator<? super E> comp)
PriorityQueue(int capacity, Comparator<? super E> comp)
PriorityQueue(Collection<? extends E> c)
PriorityQueue(PriorityQueue<? extends E> c)
PriorityQueue(SortedSet<? extends E> c)
```

Первый конструктор создает пустую очередь с начальной емкостью 11 элементов. Второй конструктор строит очередь с заданной начальной емкостью. В третьем конструкторе указывается компаратор. Четвертый конструктор строит очередь с заданной емкостью и компаратором. Последние три конструктора создают очереди, которые инициализируются элементами коллекции, переданной в c. Во всех случаях емкость увеличивается автоматически по мере добавления элементов.

Если при создании экземпляра PriorityQueue компаратор не указан, тогда применяется стандартный компаратор для типа данных, хранящихся в очереди. Стандартный компаратор упорядочивает очередь в порядке возрастания. Таким образом, в голове очереди будет наименьшее значение. Однако за счет предоставления специального компаратора появляется возможность указания другой схемы упорядочения. Скажем, при хранении элементов, содержащих отметку времени, можно установить приоритет очереди, чтобы самые старые элементы находились в очереди первыми.

Получить ссылку на компаратор, используемый экземпляром PriorityQueue, можно с помощью его метода comparator():

```
Comparator<? super E> comparator()
```

Метод возвращает компаратор. В случае применения в вызывающей очереди естественного упорядочивания возвращается null.

Одно предостережение: несмотря на возможность прохода по PriorityQueue с использованием итератора, порядок итерации не определен. Для корректной работы с PriorityQueue потребуется вызывать такие методы, как offer() и poll(), которые определены в интерфейсе Queue.

Класс ArrayDeque

Класс ArrayDeque расширяет AbstractCollection и реализует интерфейс Deque. Никакие собственные методы не добавляются. ArrayDeque создает динамический массив и не имеет ограничений по емкости. (Интерфейс Deque поддерживает реализации, которые ограничивают емкость, но не требует таких ограничений.) ArrayDeque — обобщенный класс, который имеет приведенное ниже объявление:

```
class ArrayDeque<E>
```

В E указывается тип объектов, хранящихся коллекции.

Вот конструкторы, которые определены в ArrayDeque:

```
ArrayDeque()
ArrayDeque(int size)
ArrayDeque(Collection<? extends E> c)
```

Первый конструктор создает пустую двустороннюю очередь с начальной емкостью 16 элементов. Второй конструктор строит очередь с указанной начальной емкостью. Третий конструктор создает двустороннюю очередь, которая инициализируется элементами коллекции, переданной в c. Во всех случа-

ях емкость увеличивается по мере необходимости для обработки элементов, добавленных в очередь.

В следующей программе демонстрируется применение класса ArrayDeque для создания стека:

```
// Демонстрация использования ArrayDeque.
import java.util.*;

class ArrayDequeDemo {
  public static void main(String[] args) {
    // Создать двустороннюю очередь в виде массива.
    ArrayDeque<String> adq = new ArrayDeque<String>();

    // Использовать экземпляр ArrayDeque подобно стеку.
    adq.push("A");
    adq.push("B");
    adq.push("D");
    adq.push("E");
    adq.push("F");

    System.out.print("Извлечение из стека: ");

    while(adq.peek() != null)
      System.out.print(adq.pop() + " ");

    System.out.println();
  }
}
```

Вот вывод программы:

```
Извлечение из стека: F E D B A
```

Класс EnumSet

Класс EnumSet расширяет AbstractSet и реализует интерфейс Set. Он предназначен для использования с объектами типа enum и является обобщенным классом с таким объявлением:

```
class EnumSet<E extends Enum<E>>
```

В E указывается тип элементов. Обратите внимание, что тип E должен расширять Enum<E>, обеспечивая соблюдение требования о принадлежности элементов к заданному типу enum.

Конструкторы в классе EnumSet не определены. Взамен для создания объектов применяются фабричные методы, кратко описанные в табл. 20.9.

Все методы могут генерировать исключение NullPointerException. Методы copyOf() и range() вдобавок способны генерировать исключение IllegalArgumentException. Обратите внимание, что метод of() перегружен несколько раз, что сделано в интересах эффективности. Передача известного количества аргументов может быть быстрее, чем использование аргументов переменной длины, когда количество аргументов невелико.

Таблица 20.9. Методы, объявленные в классе EnumSet

Метод	Описание
`static <E extends Enum<E>> EnumSet<E> allOf(Class<E> t)`	Создает экземпляр EnumSet, который содержит элементы из перечисления, указанного в t
`static <E extends Enum<E>> EnumSet<E> complementOf (EnumSet<E> e)`	Создает экземпляр EnumSet, который состоит из элементов, отсутствующих в e
`static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> c)`	Создает экземпляр EnumSet, который состоит из элементов, хранящихся в c
`static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)`	Создает экземпляр EnumSet, который состоит из элементов, хранящихся в c
`static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> t)`	Создает экземпляр EnumSet с элементами, отсутствующими в перечислении t, которое по определению является пустым набором
`static <E extends Enum<E>> EnumSet<E> of(E v, E ... varargs)`	Создает экземпляр EnumSet, который содержит элементы из v и ноль или больше дополнительных значений перечисления
`static <E extends Enum<E>> EnumSet<E> of(E v)`	Создает экземпляр EnumSet, который содержит элементы из v
`static <E extends Enum<E>> EnumSet<E> of(E v1, E v2)`	Создает экземпляр EnumSet, который содержит элементы из v1 и v2
`static <E extends Enum<E>> EnumSet<E> of(E v1, E v2, E v3)`	Создает экземпляр EnumSet, который содержит элементы из v1–v3
`static <E extends Enum<E>> EnumSet<E> of(E v1, E v2, E v3, E v4)`	Создает экземпляр EnumSet, который содержит элементы из v1–v4
`static <E extends Enum<E>> EnumSet<E> of(E v1, E v2, E v3, E v4, E v5)`	Создает экземпляр EnumSet, который содержит элементы из v1–v5
`static <E extends Enum<E>> EnumSet<E> range(E start, E end)`	Создает экземпляр EnumSet, который содержит элементы в диапазоне, указанном с помощью start и end

Доступ в коллекцию через итератор

Потребность в проходе по элементам коллекции будет возникать часто. Например, задача может заключаться в отображении всех элементов. Один из способов ее решения предусматривает применение *итератора*, который представляет собой объект, реализующий интерфейс `Iterator` или `ListIterator`. Итератор позволяет циклически проходить по коллекции, получая или удаляя элементы. Интерфейс `ListIterator` расширяет `Iterator` с целью обеспечения двунаправленного обхода списка и модификации элементов. `Iterator` и `ListIterator` являются обобщенными интерфейсами со следующими объявлениями:

```
interface Iterator<E>
interface ListIterator<E>
```

В `E` указывается тип объектов, по которым будет осуществляться проход. В интерфейсе `Iterator` объявлены методы, кратко описанные в табл. 20.10. Методы, объявленные в `ListIterator` (вместе с теми, что унаследованы от `Iterator`), показаны в табл. 20.11. В обоих случаях операции, изменяющие базовую коллекцию, являются необязательными. Скажем, метод `remove()` сгенерирует исключение `UnsupportedOperationException` при использовании с коллекцией, доступной только для чтения. Возможно возникновение и ряда других исключений.

На заметку! Для циклического прохода по коллекции можно также применять интерфейс `Spliterator`, который работает иначе, чем `Iterator`, и рассматривается позже в этой главе.

Таблица 20.10. Методы, объявленные в интерфейсе `Iterator`

Метод	Описание
`default void forEachRemaining (Consumer<? super E> action)`	Выполняет действие, указанное в `action`, для каждого необработанного элемента в коллекции
`boolean hasNext()`	Возвращает `true`, если в коллекции еще остались элементы, или `false` в противном случае
`E next()`	Возвращает следующий элемент. Генерирует исключение `NoSuchElementException`, если следующий элемент отсутствует
`default void remove()`	Удаляет текущий элемент. Генерирует исключение `IllegalStateException`, если предпринята попытка вызова `remove()`, которому не предшествовал вызов `next()`. Стандартная версия генерирует исключение `UnsupportedOperationException`

Таблица 20.11. Методы, объявленные в интерфейсе `ListIterator`

Метод	Описание
`void add(E obj)`	Вставляет `obj` в список перед элементом, который будет возвращен следующим вызовом метода `next()`
`default void forEachRemaining (Consumer<? super E> action)`	Выполняет действие, указанное в `action`, для каждого необработанного элемента в коллекции
`boolean hasNext()`	Возвращает `true` при наличии следующего элемента или `false` в противном случае
`boolean hasPrevious()`	Возвращает `true` при наличии предыдущего элемента или `false` в противном случае
`E next()`	Возвращает следующий элемент. Генерирует исключение `NoSuchElementException`, если следующий элемент отсутствует
`int nextIndex()`	Возвращает индекс следующего элемента или размер списка, если следующий элемент отсутствует
`E previous()`	Возвращает предыдущий элемент. Генерирует исключение `NoSuchElementException`, если предыдущий элемент отсутствует
`int previousIndex()`	Возвращает индекс предыдущего элемента или `-1`, если предыдущий элемент отсутствует
`void remove()`	Удаляет из списка текущий элемент. Генерирует исключение `IllegalStateException`, если метод `remove()` был вызван до вызова `next()` или `previous()`
`void set(E obj)`	Присваивает `obj` текущему элементу, который представляет собой элемент, возвращенный вызовом `next()` или `previous()`

Использование итератора

Чтобы появилась возможность доступа к коллекции через итератор, его необходимо получить. Каждый из классов коллекций предоставляет метод `iterator()`, который возвращает итератор для начала коллекции. С применением такого объекта итератора можно обращаться к каждому элементу коллекции по одному за раз. В общем случае для использования итератора с целью циклического просмотра содержимого коллекции понадобится выполнить описанные ниже действия.

1. Получить итератор для начала коллекции, вызвав метод `iterator()` коллекции.
2. Организовать цикл, внутри которого вызывается метод `hasNext()`. Обеспечить выполнение цикла, пока `hasNext()` возвращает true.
3. Получать в цикле элементы с помощью вызова `next()`.

Для коллекций, реализующих интерфейс `List`, получить итератор можно также, вызвав метод `listIterator()`. Как объяснялось ранее, списковый итератор обеспечивает возможность доступа к коллекции в прямом или обратном направлении и позволяет модифицировать элемент. В остальном `ListIterator` применяется точно так же, как `Iterator`.

Перечисленные выше шаги демонстрируются в следующем примере, где задействованы интерфейсы `Iterator` и `ListIterator`. В нем используется объект `ArrayList`, но общие принципы применимы к коллекциям любого типа. Конечно, итератор `ListIterator` доступен только для тех коллекций, которые реализуют интерфейс `List`.

```
// Демонстрация работы итераторов.
import java.util.*;
class IteratorDemo {
  public static void main(String[] args) {
    // Создать списковый массив.
    ArrayList<String> al = new ArrayList<String>();
    // Добавить элементы в списковый массив.
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");
    // Использовать итератор для отображения содержимого al.
    System.out.print("Исходное содержимое al: ");
    Iterator<String> itr = al.iterator();
    while(itr.hasNext()) {
      String element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();
    // Модифицировать объекты в ходе итерации.
    ListIterator<String> litr = al.listIterator();
    while(litr.hasNext()) {
      String element = litr.next();
      litr.set(element + "+");
    }
    System.out.print("Модифицированное содержимое al: ");
    itr = al.iterator();
    while(itr.hasNext()) {
      String element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();
```

```
    // Отобразить список в обратном порядке.
    System.out.print("Модифицированный список в обратном порядке: ");
    while(litr.hasPrevious()) {
      String element = litr.previous();
      System.out.print(element + " ");
    }
    System.out.println();
  }
}
```

Вот вывод, генерируемый программой:

```
Исходное содержимое al: C A E B D F
Модифицированное содержимое al: C+ A+ E+ B+ D+ F+
Модифицированный список в обратном порядке: F+ D+ B+ E+ A+ C+
```

Обратите особое внимание на способ отображения списка в обратном порядке. После изменения списка `litr` указывает на конец списка. (Вспомните, что `litr.hasNext()` возвращает `false`, когда достигнут конец списка.) Для прохода по списку в обратном порядке программа продолжает использовать `litr`, но на этот раз предпринимается проверка, существует ли предыдущий элемент. До тех пор, пока предыдущий элемент присутствует, он получается и отображается.

Альтернатива итераторам в виде цикла `for` в стиле "for-each"

Если вы не планируете модифицировать содержимое коллекции или получать элементы в обратном порядке, тогда цикл `for` в стиле "for-each" часто будет более удобной альтернативой циклическому проходу по коллекции, нежели применение итератора. Как вам уже известно, цикл `for` может циклически проходить по коллекции объектов, реализующей интерфейс `Iterable`. Поскольку интерфейс `Iterable` реализуют все классы коллекций, все они способны работать с циклом `for`.

В следующем примере цикл `for` используется для суммирования элементов коллекции:

```
// Использование цикла for в стиле "for-each" для прохода по коллекции.
import java.util.*;

class ForEachDemo {
  public static void main(String[] args) {
    // Создать списковый массив для целых чисел.
    ArrayList<Integer> vals = new ArrayList<Integer>();

    // Добавить значения в списковый массив.
    vals.add(1);
    vals.add(2);
    vals.add(3);
    vals.add(4);
    vals.add(5);
```

```
    // Использовать цикл for для отображения значений.
    System.out.print("Содержимое vals: ");
    for(int v : vals)
      System.out.print(v + " ");
    System.out.println();
    // Просуммировать значения с применением цикла for.
    int sum = 0;
    for(int v : vals)
      sum += v;
    System.out.println("Сумма значений: " + sum);
  }
}
```

Программа генерирует такой вывод:

```
Содержимое vals: 1 2 3 4 5
Сумма значений: 15
```

Как видите, цикл `for` значительно короче и проще подхода, основанного на итераторе. Тем не менее, его можно применять только для циклического прохода по коллекции в прямом направлении, но не изменять содержимое коллекции.

Сплитераторы

В версии JDK 8 появился еще один тип итератора под названием *сплитератор*, который определен интерфейсом `Spliterator`. Сплитератор проходит в цикле по последовательности элементов и в этом отношении он подобен описанным ранее итераторам. Однако приемы, необходимые для его использования, различаются. Кроме того, сплитератор предлагает значительно больше функциональных возможностей, нежели `Iterator` или `ListIterator`. Возможно, самым важным аспектом интерфейса `Spliterator` является его способность поддерживать параллельную итерацию частей последовательности. Таким образом, `Spliterator` поддерживает параллельное программирование. (Информацию о параллелизме и параллельном программировании ищите в главе 29.) Тем не менее, реализацию интерфейса `Spliterator` можно применять, даже если не планируется использовать параллельное выполнение. Одна из причин заключается в том, что `Spliterator` предлагает оптимизированный подход, который объединяет операции, реализуемые методами `hasNext()` и `next()`, в один метод.

`Spliterator` — обобщенный интерфейс со следующим объявлением:

```
interface Spliterator<T>
```

В `T` указывается тип элементов, по которым будет осуществляться проход. Методы, объявленные в `Spliterator`, кратко описаны в табл. 20.12.

Таблица 20.12. Методы, объявленные в интерфейсе `Spliterator`

Метод	Описание
`int characteristics()`	Возвращает характеристики вызывающего сплитератора, закодированные в виде целого значения
`long estimateSize()`	Оценивает количество элементов, оставшихся для итерации, и возвращает результат. Возвращает `Long.MAX_VALUE`, если по какой-либо причине счетчик не может быть получен
`default void forEachRemaining (Consumer<? super T> action)`	Применяет действие, указанное в `action`, к каждому необработанному элементу в источнике данных
`default Comparator <? super T> getComparator()`	Возвращает компаратор, используемый вызывающим сплитератором, или `null` в случае применения естественного упорядочения. Если последовательность не упорядочена, тогда генерируется исключение `IllegalStateException`
`default long getExactSizeIfKnown()`	Возвращает количество элементов, которые осталось подвергнуть итерации, если для вызывающего сплитератора установлен размер, или `-1` в противном случае
`default boolean hasCharacteristics (int val)`	Возвращает `true`, если вызывающий сплитератор имеет характеристики, переданные в `val`, или `false` в противном случае
`boolean tryAdvance (Consumer<? super T> action)`	Выполняет действие, указанное в `action`, над следующим элементом в итерации. Возвращает `true` при наличии следующего элемента или `false`, если элементов больше не осталось
`Spliterator<T> trySplit()`	По возможности разбивает на части вызывающий сплитератор и возвращает ссылку на новый сплитератор для части или `null`, если разбиение выполнить не удалось. Таким образом, в случае успеха исходный сплитератор проходит по одной части последовательности, а возвращаемый сплитератор — по другой

Использовать интерфейс Spliterator для базовых итерационных задач довольно просто: нужно лишь вызывать метод tryAdvance(), пока он не возвратит false. В случае применения одного и того же действия к каждому элементу последовательности метод forEachRemaining() предлагает облегченную альтернативу. В обеих ситуациях действие, которое будет происходить при каждой итерации, определяется тем, что именно объект Consumer делает с каждым элементом. Применение действия к объекту обеспечивает обобщенный функциональный интерфейс Consumer, который объявлен в пакете java.util.function. (Информацию о java.util.function ищите в главе 21.) В Consumer имеется только один абстрактный метод accept():

```
void accept(T objRef)
```

В случае метода tryAdvance() каждая итерация передает следующий элемент последовательности в objRef. Часто самый простой способ реализации интерфейса Consumer предусматривает использование лямбда-выражения.

В следующей программе демонстрируется простой пример работы с интерфейсом Spliterator. Обратите внимание, что в ней задействован как метод tryAdvance(), так и метод forEachRemaining(). Вдобавок взгляните на то, как эти методы объединяют действия методов next() и hasNext() из Iterator в один вызов.

```
// Простая демонстрация использования Spliterator.
import java.util.*;

class SpliteratorDemo {
  public static void main(String[] args) {
    // Создать списковый массив для элементов типа double.
    ArrayList<Double> vals = new ArrayList<>();
    // Добавить значения в списковый массив.
    vals.add(1.0);
    vals.add(2.0);
    vals.add(3.0);
    vals.add(4.0);
    vals.add(5.0);
    // Использовать tryAdvance() для отображения содержимого vals.
    System.out.print("Содержимое vals:\n");
    Spliterator<Double> spltitr = vals.spliterator();
    while(spltitr.tryAdvance((n) -> System.out.println(n)));
    System.out.println();
    // Создать новый список, содержащий значения квадратных
    // корней элементов из vals.
    spltitr = vals.spliterator();
    ArrayList<Double> sqrs = new ArrayList<>();
    while(spltitr.tryAdvance((n) -> sqrs.add(Math.sqrt(n))));
    // Использовать forEachRemaining() для отображения содержимого sqrs.
    System.out.print("Содержимое sqrs:\n");
    spltitr = sqrs.spliterator();
    spltitr.forEachRemaining((n) -> System.out.println(n));
    System.out.println();
  }
}
```

Вот вывод, генерируемый программой:

```
Содержимое vals:
1.0
2.0
3.0
4.0
5.0

Содержимое sqrs:
1.0
1.4142135623730951
1.7320508075688772
2.0
2.23606797749979
```

Хотя в приведенной выше программе демонстрировался способ использования интерфейса Spliterator, там не была раскрыта вся его мощь. Как уже упоминалось, максимальное преимущество интерфейс Spliterator проявляет в ситуациях, связанных с параллельной обработкой.

Обратите особое внимание на методы characteristics() и hasCharacteristics(), описанные в табл. 20.12. Каждый экземпляр реализации Spliterator имеет набор ассоциированных с ним атрибутов, которые называются *характеристиками*. Они определяются в Spliterator с помощью статических полей типа int, таких как SORTED, DISTINCT, SIZED и IMMUTABLE — и это лишь несколько из них. Получить характеристики можно, вызвав метод characteristics(). Чтобы выяснить, присутствует ли характеристика, необходимо вызвать метод hasCharacteristics(). Доступ к характеристикам Spliterator часто не будет нужен, но в некоторых случаях они могут помочь в создании эффективного и отказоустойчивого кода.

На заметку! Рассмотрение интерфейса Spliterator продолжается в главе 30, где он используется в контексте потокового API. Лямбда-выражения обсуждались в главе 15, а параллельное программирование и параллелизм будут описаны в главе 29.

Есть несколько вложенных интерфейсов Spliterator, которые предназначены для использования с примитивными типами double, int и long и называются Spliterator.OfDouble, Spliterator.OfInt и Spliterator.OfLong. Существует также обобщенная версия Spliterator.OfPrimitive(), которая обеспечивает дополнительную гибкость и служит суперинтерфейсом для ранее упомянутых интерфейсов.

Хранение объектов пользовательских классов в коллекциях

Для простоты в приведенных выше примерах в коллекции сохранялись встроенные объекты вроде String либо Integer. Разумеется, коллекции не ограничиваются возможностью быть хранилищем только для встроенных объектов. Наоборот, мощь коллекций заключается в том, что они способны

хранить объекты любого типа, в том числе объекты классов, создаваемых пользователем. Например, рассмотрим следующий пример, в котором используется LinkedList для хранения почтовых адресов:

```
// Простой пример работы со списком почтовых адресов.
import java.util.*;

class Address {
  private String name;
  private String street;
  private String city;
  private String state;
  private String code;
  Address(String n, String s, String c,
          String st, String cd) {
    name = n;
    street = s;
    city = c;
    state = st;
    code = cd;
  }

  public String toString() {
    return name + "\n" + street + "\n" + city + " " + state + " " + code;
  }
}
class MailList {
  public static void main(String[] args) {
    LinkedList<Address> ml = new LinkedList<Address>();

    // Добавить элементы в связный список.
    ml.add(new Address("J.W. West", "11 Oak Ave",
                       "Urbana", "IL", "61801"));
    ml.add(new Address("Ralph Baker", "1142 Maple Lane",
                       "Mahomet", "IL", "61853"));
    ml.add(new Address("Tom Carlton", "867 Elm St",
                       "Champaign", "IL", "61820"));
    // Отобразить содержимое списка почтовых адресов.
    for(Address element : ml)
      System.out.println(element + "\n");

    System.out.println();
  }
}
```

Ниже показан вывод, генерируемый программой:

```
J.W. West
11 Oak Ave
Urbana IL 61801

Ralph Baker
1142 Maple Lane
Mahomet IL 61853

Tom Carlton
867 Elm St
Champaign IL 61820
```

Помимо сохранения определяемого пользователем класса в коллекции есть еще один важный момент, который следует отметить в предыдущей программе — она довольно короткая. Если учесть, что она настраивает связный список, который способен сохранять, извлекать и обрабатывать почтовые адреса примерно в 50 строках кода, то мощь инфраструктуры Collections Framework становится вполне очевидной. Как известно большинству читателей, если бы всю эту функциональность пришлось кодировать вручную, то программа оказалась бы в несколько раз длиннее. Коллекции предлагают готовые решения для самых разных задач программирования. Вы должны использовать их всякий раз, когда возникает подходящая ситуация.

Интерфейс RandomAccess

Интерфейс `RandomAccess` не содержит членов. Однако за счет реализации данного интерфейса коллекция сигнализирует о том, что она поддерживает эффективный произвольный доступ к своим элементам. Хотя коллекция способна поддерживать произвольный доступ, она может оказаться неэффективной. Проверяя реализацию интерфейса `RandomAccess`, клиентский код может определить во время выполнения, подходит ли коллекция для определенных типов операций произвольного доступа, особенно если они применяются к большим коллекциям. (Для выяснения, реализует ли класс какой-либо интерфейс, можно использовать `instanceof`.) Интерфейс `RandomAccess` реализуется среди прочих классом `ArrayList` и унаследованным классом `Vector`.

Работа с картами

Карта представляет собой объект, в котором хранятся ассоциации между ключами и значениями, или *пары "ключ-значение"*. Имея ключ, вы можете найти его значение. И ключи, и значения являются объектами. Ключи должны быть уникальными, но значения могут дублироваться. Одни карты могут принимать ключ `null` и значения `null`, другие же нет.

С картами связан один момент, о котором важно упомянуть в самом начале: они не реализуют интерфейс `Iterable`. Это означает *невозможность* прохода по карте с применением цикла `for` в стиле "for-each". Кроме того, нельзя получить итератор для карты. Тем не менее, как вскоре будет показано, можно получить представление карты в виде коллекции, которое позволяет использовать либо цикл `for`, либо итератор.

Интерфейсы карт

Поскольку интерфейсы карт определяют характер и природу карт, именно с них начинается обсуждение карт. Карты поддерживаются интерфейсами, перечисленными в табл. 20.13.

Таблица 20.13. Интерфейсы, которые поддерживают карты

Интерфейс	Описание
`Map`	Сопоставляет уникальные ключи со значениями
`Map.Entry`	Описывает элемент (пару "ключ-значение") в карте. Является вложенным интерфейсом `Map`
`NavigableMap`	Расширяет интерфейс `SortedMap` для поддержки извлечения элементов на основе поиска с наиболее точным совпадением
`SortedMap`	Расширяет интерфейс `Map`, чтобы ключи сохранялись в возрастающем порядке

Давайте рассмотрим все интерфейсы по очереди.

Интерфейс Map

Интерфейс `Map` сопоставляет уникальные ключи со значениями. Ключ — это объект, который используется для извлечения значения в будущем. Имея ключ и значение, это значение можно сохранить в объекте `Map`. После сохранения получить значение можно с помощью ключа. Интерфейс `Map` является обобщенным и объявляется, как показано ниже:

```
interface Map<K, V>
```

В `K` указывается тип ключей, а в `V` — тип значений.

Методы, объявленные в интерфейсе `Map`, кратко описаны в табл. 20.14. Некоторые методы генерируют исключение `ClassCastException`, когда объект оказывается несовместимым с элементами в карте.

Таблица 20.14. Методы, объявленные в интерфейсе Map

Метод	Описание
`void clear()`	Удаляет все пары "ключ-значение" из вызывающей карты
`default V compute(K k, BiFunction<? super K, ? super V, ? extends V> func)`	Вызывает функцию, указанную в `func`, для конструирования нового значения. Если `func` возвращает значение, отличающееся от `null`, тогда в карту добавляется новая пара "ключ-значение", любая ранее существовавшая пара удаляется и возвращается новое значение. Если `func` возвращает `null`, то любая ранее существовавшая пара удаляется, после чего возвращается `null`

Продолжение табл. 20.14

Метод	Описание
`default V computeIfAbsent(K k, Function<? super K, ? extends V> func)`	Возвращает значение, ассоциированное с ключом k. В противном случае через вызов func конструируется значение, в карту добавляется новая пара и возвращается сконструированное значение. Если значение не может быть сконструировано, тогда возвращается null
`default V computeIfPresent(K k, BiFunction<? super K, ? super V,? extends V> func)`	Если ключ k присутствует в карте, то через вызов func конструируется новое значение, которое заменяет собой старое значение в карте. В этом случае возвращается новое значение. Если func возвращает null, тогда существующий ключ и значение удаляются из карты и возвращается null
`boolean containsKey(Object k)`	Возвращает true, если вызывающая карта содержит k в качестве ключа, или false в противном случае
`boolean containsValue(Object v)`	Возвращает true, если вызывающая карта содержит v в качестве значения, или false в противном случае
`static <K,V> Map<K,V> copyOf(Map<? extends K, ? extends V> from)`	Возвращает карту, которая содержит такие же пары "ключ-значение", как карта, указанная в from. Возвращаемая карта является неизменяемой и основанной на значении. Ключи или значения null не разрешены
`static <K, V> Map.Entry<K, V> entry(K k, V v)`	Возвращает неизменяемый и основанный на значении элемент карты, образованный из указанного ключа и значения. Ключ или значение null не разрешено
`Set<Map.Entry<K, V>> entrySet()`	Возвращает объект Set с элементами карты. Набор содержит объекты типа Map.Entry. Таким образом, этот метод возвращает представление в виде набора вызывающей карты
`boolean equals (Object obj)`	Возвращает true, если obj является объектом Map и содержит те же самые элементы, или false в противном случае
`default void forEach (BiConsumer<? super K, ? super V> action)`	Выполняет действие, указанное в action, над каждым элементом в вызывающей карте. Если в процессе какой-то элемент удаляется, тогда будет сгенерировано исключение ConcurrentModificationException

Продолжение табл. 20.14

Метод	Описание
`V get(Object k)`	Возвращает значение, ассоциированное с ключом k, или null, если ключ не найден
`default V getOrDefault (Object k, V defVal)`	Возвращает значение, ассоциированное с ключом k, если он присутствует в карте, и defVal в противном случае
`int hashCode()`	Возвращает хеш-код для вызывающей карты
`boolean isEmpty()`	Возвращает true, если вызывающая карта пуста, или false в противном случае
`Set<K> keySet()`	Возвращает объект Set с ключами вызывающей карты. Этот метод предоставляет представление в виде набора для ключей вызывающей карты
`default V merge(K k, V v, BiFunction<? super V, ? super V, ? extends V> func)`	Если ключ k в карте отсутствует, тогда в карту добавляется пара k, v и возвращается v. В противном случае func возвращает новое значение на основе старого, ключ обновляется с использованием этого значения, которое и возвращает метод merge(). Если func возвращает null, тогда существующий ключ и значение удаляются из карты и возвращается null
`static <K, V> Map<K, V> of(`*список-параметров*`)`	Создает неизменяемую и основанную на значении карту с элементами, указанными в списке параметров. Ключи или значения null не разрешены. Существует много перегруженных версий, которые более подробно обсуждаются далее в книге
`static <K, V> Map<K, V> ofEntries(Map.Entry <? extends K, ? extends V>...entries)`	Возвращает неизменяемую и основанную на значении карту с парами "ключ-значение", описанными с помощью элементов, которые переданы в entries. Ключи или значения null не разрешены
`V put(K k, V v)`	Помещает элемент в вызывающую карту с перезаписыванием предыдущего значения, ассоциированного с ключом. Ключ и значение указываются соответственно в k и v. Возвращает null, если ключ не существует, или предыдущее значение, связанное с ключом, в противном случае

Окончание табл. 20.14

Метод	Описание
`void putAll(Map<? extends K, ? extends V> m)`	Помещает все элементы из `m` в вызывающую карту
`default V putIfAbsent (K k, V v)`	Вставляет пару "ключ-значение" в вызывающую карту, если такая пара еще не присутствует или существующее значение равно `null`. Возвращает старое значение. Если пара отсутствует или значение равно `null`, тогда возвращается `null`
`V remove(Object k)`	Удаляет элемент, ключ которого равен `k`
`default boolean remove(Object k, Object v)`	Если пара "ключ-значение", указанная с помощью `k` и `v`, присутствует в вызывающей карте, тогда она удаляется и возвращается `true`. В противном случае возвращается `false`
`default boolean replace(K k, V oldV, V newV)`	Если пара "ключ-значение", указанная с помощью `k` и `oldV`, присутствует в вызывающей карте, тогда значение заменяется `newV` и возвращается `true`. В противном случае возвращается `false`
`default V replace (K k, V v)`	Если ключ, указанный в `k`, присутствует в вызывающей карте, тогда его значение устанавливается в `v` и возвращается предыдущее значение. В противном случае возвращается `null`
`default void replaceAll(BiFunction <? super K, ? super V, ? extends V> func)`	Выполняет `func` с каждым элементом вызывающей карты, заменяя элемент результатом, который возвращает `func`. Если в процессе какой-то элемент удаляется, тогда будет сгенерировано исключение `ConcurrentModificationException`
`int size()`	Возвращает количество пар "ключ-значение" в карте
`Collection<V> values()`	Возвращает коллекцию, содержащую значения в карте. Этот метод предоставляет представление в виде набора для значений вызывающей карты

При попытке использования объекта null, а значения null в карте не разрешены, генерируется исключение NullPointerException. При попытке внесения изменений в неизменяемую карту генерируется исключение UnsupportedOperationException. В случае применения недопустимого аргумента генерируется исключение IllegalArgumentException.

Карты вращаются вокруг двух базовых операций: get() и put(). Чтобы поместить значение в карту, используйте put(), указав ключ и значение. Чтобы получить значение, вызывайте get(), передав ключ в качестве аргумента. Значение возвращается.

Как упоминалось ранее, хотя карты входят в состав инфраструктуры Collections Framework, сами по себе они не являются коллекциями, поскольку не реализуют интерфейс Collection. Однако можно получить представление в виде коллекции для карты с помощью метода entrySet(), который возвращает набор, содержащий элементы карты. Чтобы получить представление в виде коллекции для ключей, применяйте keySet(). Чтобы получить представление в виде коллекции для значений, используйте values(). Для всех трех представлений в виде коллекций коллекция поддерживается в актуальном состоянии картой. Изменение одного представления влияет на другое. Представления в виде коллекций — это средства, с применением которых карты интегрируются в более крупную инфраструктуру Collections Framework.

Начиная с версии JDK 9, интерфейс Map включает фабричный метод of(), который имеет несколько перегруженных версий. Каждая версия возвращает неизменяемую карту на основе значений, образованную из переданных аргументов. Основная цель of() — предложить удобный и эффективный способ создания небольшой карты. Существует 11 перегруженных версий of(). Одна из них не принимает аргументов и создает пустую карту:

```
static <K, V> Map<K, V> of()
```

Десять перегруженных версий принимают от 1 до 10 аргументов и создают список, содержащий указанные элементы:

```
static <K, V> Map<K, V> of(K k1, V v1)
static <K, V> Map<K, V> of(K k1, V v1, K k2, V v2)
static <K, V> Map<K, V> of(K k1, V v1, K k2, V v2, K k3, V v3)
...
static <K, V> Map<K, V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4,
  K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9, K k10, V v10)
```

Во всех версиях ключи и/или значения, равные null, не разрешены. Во всех случаях реализация Map не указывается.

Интерфейс SortedMap

Интерфейс SortedMap расширяет Map. Он гарантирует, что элементы поддерживаются в возрастающем порядке на основе ключей. Интерфейс SortedMap является обобщенным и имеет следующее объявление:

```
interface SortedMap<K, V>
```

В K указывается тип ключей, а в V — тип значений.

Методы, добавленные интерфейсом SortedMap, кратко описаны в табл. 20.15. Несколько методов генерируют исключение NoSuchElement Exception, если в вызывающей карте нет элементов.

Исключение ClassCastException возникает, когда объект несовместим с элементами карты. Исключение NullPointerException происходит при попытке использования объекта null, когда значения null в карте не разрешены. Исключение IllegalArgumentException генерируется в случае применения недопустимого аргумента.

Таблица 20.15. Методы, объявленные в интерфейсе SortedMap

Метод	Описание
`Comparator <? super K> comparator()`	Возвращает компаратор вызывающей отсортированной карты или null, если для вызывающей карты используется естественное упорядочение
`K firstKey()`	Возвращает первый ключ в вызывающей карте
`SortedMap<K, V> headMap(K end)`	Возвращает отсортированную карту для элементов, ключи которых меньше end
`K lastKey()`	Возвращает последний ключ в вызывающей карте
`SortedMap<K, V> subMap(K start, K end)`	Возвращает карту, содержащую элементы, ключи которых больше или равны start и меньше end
`SortedMap<K, V> tailMap(K start)`	Возвращает карту, содержащую элементы, ключи которых больше или равны start

Отсортированные карты позволяют очень эффективно манипулировать подкартами (другими словами, поднаборами карты). Для получения подкарты используйте headMap(), tailMap() или subMap(). Подкарта, возвращаемая этими методами, поддерживается в актуальном состоянии вызывающей картой. Изменение одной подкарты изменяет другую. Чтобы получить первый ключ в наборе, вызывайте firstKey(). Чтобы получить последний ключ, применяйте lastKey().

Интерфейс NavigableMap

Интерфейс NavigableMap расширяет SortedMap и обеспечивает поведение карты, которая поддерживает извлечение элементов на основе поиска с наиболее точным совпадением с заданным ключом или ключами. NavigableMap — это обобщенный интерфейс с показанным ниже объявлением:

```
interface NavigableMap<K,V>
```

В K указывается тип ключей, а в V — тип значений, ассоциированных с ключами. В дополнение к методам, унаследованным от SortedMap, в интер-

фейсе NavigableMap определены методы, кратко описанные в табл. 20.16. Некоторые методы генерируют исключение ClassCastException, когда объект несовместим с ключами в карте. Исключение NullPointerException происходит при попытке использования объекта null, а ключи null в карте не разрешены. Исключение IllegalArgumentException возникает в случае применения недопустимого аргумента.

Таблица 20.16. Методы, объявленные в интерфейсе NavigableMap

Метод	Описание
`Map.Entry<K,V> ceilingEntry (K obj)`	Ищет в карте наименьший ключ k, который больше или равен obj. Если такой ключ найден, то возвращается его элемент, а если нет, тогда возвращается null
`K ceilingKey (K obj)`	Ищет в карте наименьший ключ k, который больше или равен obj. Если такой ключ найден, то он возвращается, а если нет, тогда возвращается null
`NavigableSet<K> descendingKeySet()`	Возвращает объект NavigableSet, содержащий ключи из вызывающей карты в обратном порядке. Таким образом, этот метод возвращает обратное представление в виде набора для ключей. Результирующий набор поддерживается в актуальном состоянии картой
`NavigableMap<K,V> descendingMap()`	Возвращает объект NavigableMap, который является обратным для вызывающей карты. Результирующая карта поддерживается в актуальном состоянии вызывающей картой
`Map.Entry<K,V> firstEntry()`	Возвращает первый элемент карты, который является элементом с наименьшим ключом
`Map.Entry<K,V> floorEntry(K obj)`	Ищет в карте наибольший ключ k, который меньше или равен obj. Если такой ключ найден, то возвращается его элемент, а если нет, тогда возвращается null
`K floorKey(K obj)`	Ищет в карте наибольший ключ k, который меньше или равен obj. Если такой ключ найден, то он возвращается, а если нет, тогда возвращается null
`NavigableMap<K,V> headMap(K upperBound, boolean incl)`	Возвращает объект NavigableMap, содержащий все элементы из вызывающей карты, ключи которых меньше upperBound. Если incl равно true, то включается элемент с ключом, равным upperBound. Результирующая карта поддерживается в актуальном состоянии вызывающей картой

Окончание табл. 20.16

Метод	Описание
`Map.Entry<K,V> higherEntry(K obj)`	Ищет в карте наибольший ключ k, который больше obj. Если такой ключ найден, то возвращается его элемент, а если нет, тогда возвращается null
`K higherKey(K obj)`	Ищет в карте наибольший ключ k, который больше obj. Если такой ключ найден, то он возвращается, а если нет, тогда возвращается null
`Map.Entry<K,V> lastEntry()`	Возвращает последний элемент карты, который является элементом с наибольшим ключом
`Map.Entry<K,V> lowerEntry(K obj)`	Ищет в карте наибольший ключ k, который меньше obj. Если такой ключ найден, то возвращается его элемент, а если нет, тогда возвращается null
`K lowerKey(K obj)`	Ищет в карте наибольший ключ k, который меньше obj. Если такой ключ найден, то он возвращается, а если нет, тогда возвращается null
`NavigableSet<K> navigableKeySet()`	Возвращает объект NavigableSet, содержащий ключи из вызывающей карты. Результирующий набор поддерживается в актуальном состоянии вызывающей картой
`Map.Entry<K,V> pollFirstEntry()`	Возвращает первый элемент, в процессе удаляя его. Поскольку карта отсортирована, это будет элемент с наименьшим значением ключа. Если карта пуста, тогда возвращается null
`Map.Entry<K,V> pollLastEntry()`	Возвращает последний элемент, в процессе удаляя его. Поскольку карта отсортирована, это будет элемент с наибольшим значением ключа. Если карта пуста, тогда возвращается null
`NavigableMap<K,V> subMap (K lowerBound, boolean lowIncl, K upperBound boolean highIncl)`	Возвращает объект NavigableMap, содержащий все элементы из вызывающей карты, ключи которых больше lowerBound и меньше upperBound. Если lowIncl равно true, то включается элемент с ключом, равным lowerBound. Если highIncl равно true, то включается элемент с ключом, равным upperBound. Результирующая карта поддерживается в актуальном состоянии вызывающей картой
`NavigableMap<K,V> tailMap (K lowerBound, boolean incl)`	Возвращает объект NavigableMap, содержащий все элементы из вызывающей карты, ключи которых больше lowerBound. Если incl равно true, то включается элемент с ключом, равным lowerBound. Результирующая карта поддерживается в актуальном состоянии вызывающей картой

Интерфейс Map.Entry

Интерфейс Map.Entry позволяет работать с элементом карты. Вспомните, что метод entrySet(), объявленный в интерфейсе Map, возвращает объект Set, содержащий элементы карты. Каждый из таких элементов набора представляет собой объект реализации Map.Entry. Интерфейс Map.Entry является обобщенным и объявлен следующим образом:

```
interface Map.Entry<K, V>
```

В K указывается тип ключей, а в V — тип значений. В табл. 20.17 приведены нестатические методы, объявленные в Map.Entry. Вдобавок есть три статических метода. Первый — CompareByKey(), который возвращает экземпляр реализации Comparator, сравнивающий элементы по ключу. Второй — compareByValue(), возвращающий экземпляр реализации Comparator, который сравнивает элементы по значению. Третий — copyOf(), появившийся в версии JDK 17, возвращает неизменяемый объект на основе значения, который является копией вызывающего объекта, но не частью карты.

Таблица 20.17. Нестатические методы, объявленные в интерфейсе Map.Entry

Метод	Описание
boolean equals(Object obj)	Возвращает true, если obj является объектом реализации Map.Entry, ключ и значение которого равны ключу и значению вызывающего объекта
K getKey()	Возвращает ключ для данного элемента карты
V getValue()	Возвращает значение для данного элемента карты
int hashCode()	Возвращает хеш-код для данного элемента карты
V setValue(V v)	Устанавливает значение для данного элемента карты в v. Если v не имеет корректный тип для карты, тогда генерируется исключение ClassCastException. При наличии проблем с v возникает исключение IllegalArgumentException. Если v равно null, а в карте значения null не разрешены, то генерируется исключение NullPointerException. Если карта не может быть изменена, тогда возникает исключение UnsupportedOperationException

Классы карт

Несколько классов предлагают реализации интерфейсов карт. Классы, которые можно использовать для карт, перечислены здесь в табл. 20.18.

Таблица 20.18. Классы, предназначенные для работы с картами

Класс	Описание
AbstractMap	Реализует бо́льшую часть интерфейса Map
EnumMap	Расширяет AbstractMap для использования с перечислимыми ключами
HashMap	Расширяет AbstractMap для применения хеш-таблицы
TreeMap	Расширяет AbstractMap для использования дерева
WeakHashMap	Расширяет AbstractMap для применения хеш-таблицы со слабыми ключами
LinkedHashMap	Расширяет HashMap, чтобы сделать возможной итерацию в порядке вставки
IdentityHashMap	Расширяет AbstractMap и использует равенство ссылок при сравнении документов

Обратите внимание, что AbstractMap является суперклассом для всех конкретных реализаций карт.

Класс WeakHashMap реализует карту, в которой применяются "слабые ключи", что позволяет подвергать сборке мусора элемент карты, когда его ключ не используется иным образом. Этот класс здесь обсуждаться не будет. Другие классы карт рассматриваются далее в главе.

Класс HashMap

Класс HashMap расширяет AbstractMap и реализует интерфейс Map. Для хранения карты он применяет хеш-таблицу, что позволяет времени выполнения операций get() и put() оставаться постоянным даже для крупных карт. HashMap — обобщенный класс со следующим объявлением:

```
class HashMap<K, V>
```

В K указывается тип ключей, а в V — тип значений.

В HashMap определены перечисленные ниже конструкторы:

```
HashMap()
HashMap(Map<? extends K, ? extends V> m)
HashMap(int capacity)
HashMap(int capacity, float fillRatio)
```

Первая форма создает стандартную хеш-карту. Вторая форма инициализирует хеш-карту, используя элементы m. Третья форма инициализирует емкость хеш-карты значением capacity. Четвертая форма инициализирует емкость и коэффициент заполнения хеш-карты с применением своих аргументов. Смысл емкости и коэффициента заполнения такой же, как в описанном ранее классе HashSet. Стандартная емкость равна 16, а стандартный коэффициент заполнения — 0.75.

Класс HashMap реализует интерфейс Map и расширяет класс AbstractMap. Никаких собственных методов он не добавляет.

Важно отметить, что хеш-карта не гарантирует порядок следования элементов. По этой причине порядок, в котором элементы добавляются в хеш-карту, не обязательно совпадает с порядком, в котором они читаются итератором.

В следующей программе демонстрируется использование HashMap для сопоставления имен клиентов с балансами банковских счетов. Обратите внимание на получение и применение представления в виде набора.

```
import java.util.*;

class HashMapDemo {
  public static void main(String[] args) {

    // Создать хеш-карту.
    HashMap<String, Double> hm = new HashMap<String, Double>();

    // Поместить элементы в карту.
    hm.put("John Doe", 3434.34);
    hm.put("Tom Smith", 123.22);
    hm.put("Jane Baker", 1378.00);
    hm.put("Tod Hall", 99.22);
    hm.put("Ralph Smith", -19.08);

    // Получить набор элементов.
    Set<Map.Entry<String, Double>> set = hm.entrySet();

    // Отобразить содержимое набора.
    for(Map.Entry<String, Double> me : set) {
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }

    System.out.println();

    // Пополнить счет клиента John Doe на 1000.
    double balance = hm.get("John Doe");
    hm.put("John Doe", balance + 1000);

    System.out.println("Новый баланс клиента John Doe: " + hm.get("John Doe"));
  }
}
```

Вот вывод программы (точный порядок может варьироваться):

```
Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Tod Hall: 99.22
Jane Baker: 1378.0

Новый баланс клиента John Doe: 4434.34
```

В программе сначала создается хеш-карта и добавляются сопоставления имен клиентов с балансами. Затем содержимое карты отображается с использованием представления в виде набора, полученного вызовом entrySet(). Ключи и значения отображаются путем вызова методов getKey() и

getValue(), определенных в Map.Entry. Обратите особое внимание на то, как на расчетный счет клиента John Doe вносится депозит. Метод put() автоматически заменяет любое ранее существовавшее значение, ассоциированное с указанным ключом, новым значением. Таким образом, после обновления расчетного счета клиента John Doe хеш-карта по-прежнему будет содержать только один счет "John Doe".

Класс TreeMap

Класс TreeMap расширяет класс AbstractMap и реализует интерфейс NavigableMap. Он позволяет создавать карты, хранящиеся в древовидной структуре. Класс TreeMap предлагает эффективные средства хранения пар "ключ-значение" в отсортированном порядке и обеспечивает быстрое извлечение. Важно отметить, что в отличие от хеш-карты древовидная карта гарантирует, что ее элементы будут отсортированы в порядке возрастания ключей. TreeMap — обобщенный класс с таким объявлением:

```
class TreeMap<K, V>
```

В K указывается тип ключей, а в V — тип значений.

Вот конструкторы, которые определены в TreeMap:

```
TreeMap()
TreeMap(Comparator<? super K> comp)
TreeMap(Map<? extends K, ? extends V> m)
TreeMap(SortedMap<K, ? extends V> sm)
```

Первая форма конструирует пустую древовидную карту, которая будет отсортирована с применением естественного порядка следования ее ключей. Вторая форма строит пустую древовидную карту, сортируемую с помощью компаратора comp. (Компараторы обсуждаются далее в главе.) Третья форма инициализирует древовидную карту элементами из m, которые будут отсортированы с использованием естественного порядка следования ключей. Четвертая форма инициализирует древовидную карту элементами sm, которые будут отсортированы в том же порядке, что и sm.

Класс TreeMap не имеет методов карт помимо тех, которые определены в интерфейсе NavigableMap и классе AbstractMap.

Ниже показана переработанная версия предыдущей программы для применения в ней TreeMap:

```
import java.util.*;

class TreeMapDemo {
  public static void main(String[] args) {

    // Создать древовидную карту.
    TreeMap<String, Double> tm = new TreeMap<String, Double>();

    // Поместить элементы в карту.
    tm.put("John Doe", 3434.34);
    tm.put("Tom Smith", 123.22);
    tm.put("Jane Baker", 1378.00);
    tm.put("Tod Hall", 99.22);
```

```
    tm.put("Ralph Smith", -19.08);

    // Получить набор элементов.
    Set<Map.Entry<String, Double>> set = tm.entrySet();

    // Отобразить содержимое набора.
    for(Map.Entry<String, Double> me : set) {
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }
    System.out.println();

    // Пополнить счет клиента John Doe на 1000.
    double balance = tm.get("John Doe");
    tm.put("John Doe", balance + 1000);

    System.out.println("Новый баланс клиента John Doe: " +
                       tm.get("John Doe"));
  }
}
```

Вывод, генерируемый программой, выглядит следующим образом:

```
Jane Baker: 1378.0
John Doe: 3434.34
Ralph Smith: -19.08
Todd Hall: 99.22
Tom Smith: 123.22

Новый баланс клиента John Doe: 4434.34
```

Обратите внимание, что TreeMap сортирует ключи. Тем не менее, в данном случае они сортируются по имени, а не по фамилии. Изменить такое поведение можно путем указания компаратора при создании карты, как вскоре будет описано.

Класс LinkedHashMap

Класс LinkedHashMap расширяет HashMap и поддерживает связный список элементов карты в том порядке, в котором они были вставлены. Это позволяет выполнять итерацию по карте в порядке вставки, т.е. при проходе по представлению в виде коллекции для LinkedHashMap элементы будут возвращаться в том порядке, в каком они вставлялись. Можно также создать объект LinkedHashMap, возвращающий свои элементы в том порядке, в котором к ним обращались в последний раз. LinkedHashMap — обобщенный класс с таким объявлением:

```
class LinkedHashMap<K, V>
```

В K указывается тип ключей, а в V — тип значений.

В классе LinkedHashMap определены перечисленные ниже конструкторы:

```
LinkedHashMap()
LinkedHashMap(Map<? extends K, ? extends V> m)
LinkedHashMap(int  capacity)
LinkedHashMap(int capacity, float fillRatio)
LinkedHashMap(int capacity, float fillRatio, boolean Order)
```

Первая форма конструирует стандартный экземпляр LinkedHashMap. Вторая форма инициализирует LinkedHashMap элементами из m. Третья форма инициализирует емкость. Четвертая форма инициализирует как емкость, так и коэффициент заполнения. Смысл емкости и коэффициента заполнения такой же, как у HashMap. Стандартная емкость равна 16, а стандартный коэффициент заполнения — 0.75. Пятая форма позволяет указать, будут элементы храниться в связном списке в порядке вставки или в порядке последнего доступа. Если Order имеет значение true, тогда используется порядок доступа, а если false, то порядок вставки.

К методам, определенным в HashMap, класс LinkedHashMap добавляет только один метод — removeEldestEntry():

```
protected boolean removeEldestEntry(Map.Entry<K, V> e)
```

Данный метод вызывается методами put() и putAll(). В e передается самый старый элемент. По умолчанию метод removeEldestEntry() возвращает false и ничего не делает, но если переопределить его, тогда LinkedHashMap можно заставить удалять самую старую запись из карты. Для этого переопределенная версия должна возвращать true. Чтобы сохранить самую старую запись, понадобится возвратить false.

Класс IdentityHashMap

Класс IdentityHashMap расширяет класс AbstractMap и реализует интерфейс Map. Он похож на HashMap, но при сравнении элементов использует равенство ссылок. IdentityHashMap — обобщенный класс со следующим объявлением:

```
class IdentityHashMap<K, V>
```

В K указывается тип ключей, а в V — тип значений. В документации по API прямо указано, что класс IdentityHashMap не предназначен для общего применения.

Класс EnumMap

Класс EnumMap расширяет класс AbstractMap и реализует интерфейс Map. Он предназначен специально для использования с ключами перечислимого типа и является обобщенным классом с таким объявлением:

```
class EnumMap<K extends Enum<K>, V>
```

В K указывается тип ключей, а в V — тип значений. Обратите внимание, что тип K обязан расширять Enum<K>, что вводит требование принадлежности ключей к перечислимому типу.

Вот конструкторы, которые определены в EnumMap:

```
EnumMap(Class<K> kType)
EnumMap(Map<K, ? extends V> m)
EnumMap(EnumMap<K, ? extends V> em)
```

Первая форма конструктора создает пустую карту EnumMap типа kType. Вторая форма конструирует карту EnumMap, содержащую те же записи, что и m. Третья форма создает карту EnumMap, инициализированную значениями из em.

Никаких собственных методов класс EnumMap не определяет.

Компараторы

Классы TreeSet и TreeMap хранят элементы в отсортированном порядке. Однако именно компаратор точно определяет, что означает "отсортированный порядок". По умолчанию упомянутые классы хранят свои элементы с применением того, что в Java называется "естественным упорядочением", которое обычно соответствует ожидаемому порядку (A перед B, 1 перед 2 и т.д.). При желании упорядочить элементы по-другому во время создания набора или карты необходимо указать компаратор, что даст возможность точно управлять тем, как элементы хранятся в отсортированных коллекциях и картах.

Comparator представляет собой обобщенный интерфейс, объявленный следующим образом:

```
interface Comparator<T>
```

В T указывается тип сравниваемых объектов.

До выхода версии JDK 8 в интерфейсе Comparator были определены только два метода: compare() и equals(). Приведенный ниже метод compare() сравнивает два элемента по порядку:

```
int compare(T obj1, T obj2)
```

Здесь obj1 и obj2 — сравниваемые объекты. Обычно метод compare() возвращает ноль, если объекты равны, положительное значение, если obj1 больше obj2, и отрицательное значение, если obj1 меньше obj2. Метод может сгенерировать исключение ClassCastException, если типы объектов несовместимы для сравнения. Реализация compare() позволяете изменить способ упорядочения объектов. Скажем, для сортировки в обратном порядке можно создать компаратор, который обращает результат сравнения.

Показанный далее метод equals() проверяет, равен ли объект вызываемому компаратору:

```
boolean equals(object obj)
```

В obj указывается объект, подлежащий проверке на равенство. Метод возвращает true, если obj и вызывающий объект являются экземплярами реализации Comparator и используют одно и то же упорядочение, или false в противном случае. Переопределять equals() вовсе не обязательно, и в большинстве простых компараторов так не делается.

На протяжении многих лет в Comparator были определены только два предыдущих метода. С выходом версии JDK 8 ситуация разительно изменилась. В JDK 8 к интерфейсу Comparator добавилась важная новая функциональность за счет применения методов со стандартной реализацией и статических методов, которые рассматриваются ниже.

С помощью метода reversed() можно получить компаратор, изменяющий упорядочение компаратора, для которого он вызывается, на противоположное:

```
default Comparator<T> reversed()
```

Метод reversed() возвращает компаратор с обратным порядком. Например, если компаратор, который использует естественное упорядочение для символов от A до Z, то компаратор с обратным порядком будет помещать B перед A, C перед B и т.д.

С методом reversed() связан метод reverseOrder():

```
static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
```

Он возвращает компаратор, который изменяет естественный порядок элементов на противоположный. И наоборот, вызвав статический метод naturalOrder() можно получить компаратор, использующий естественный порядок:

```
static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
```

Если необходим компаратор, способный обрабатывать значения null, тогда подойдут методы nullsFirst() и nullsLast():

```
static <T> Comparator<T> nullsFirst(Comparator<? super T> comp)
static <T> Comparator<T> nullsLast(Comparator<? super T> comp)
```

Метод nullsFirst() возвращает компаратор, который трактует значения null как меньшие, чем другие значения. Метод nullsLast() возвращает компаратор, который трактует значения null как большие, чем другие значения. В том и другом случае, если два сравниваемых значения не равны null, то сравнение выполняет comp. Если в comp передается null, тогда все значения, отличающиеся от null, считаются эквивалентными.

Другим стандартным методом является thenComparing(). Он возвращает компаратор, который выполняет второе сравнение, когда результат первого сравнения указывает, что сравниваемые объекты равны. Таким образом, его можно применять для создания последовательности вида "сравнить по X, затем сравнить по Y". Скажем, при сравнении городов первое сравнение может иметь дело с их названиями, а второе — с их штатами. (Поэтому "Springfield, Illinois" будет располагаться перед "Springfield, Missouri" при условии нормального алфавитного порядка.) Метод thenComparing() имеет три формы. Первая позволяет указывать второй компаратор, передавая экземпляр реализации Comparator:

```
default Comparator<T> thenComparing(Comparator<? super T> thenByComp)
```

В thenByComp указывается компаратор, который вызывается, если первое сравнение возвращает признак равенства.

Следующие версии thenComparing() позволяют указывать стандартный функциональный интерфейс Function (определенный в java.util.function):

```
default <U extends Comparable<? super U> Comparator<T>
  thenComparing(Function<? super T, ? extends U> getKey)
default <U> Comparator<T>
  thenComparing(Function<? super T, ? extends U> getKey,
                Comparator<? super U> keyComp)
```

В обоих случаях getKey ссылается на функцию, получающую следующий ключ сравнения, который используется, если первое сравнение возвращает признак равенства. Во второй версии в keyComp указывается компаратор, применяемый для сравнения ключей. (Здесь и в дальнейшем U задает тип ключа.)

В интерфейсе Comparator также добавляются следующие специализированные версии методов последующего сравнения для примитивных типов:

```
default Comparator<T> thenComparingDouble(ToDoubleFunction<? super T> getKey)
default Comparator<T> thenComparingInt(ToIntFunction<? super T> getKey)
default Comparator<T> thenComparingLong(ToLongFunction<? super T> getKey)
```

Во всех методах getKey ссылается на функцию, которая получает следующий ключ для сравнения.

Наконец, интерфейс Comparator имеет метод по имени comparing(). Он возвращает компаратор, который получает свой ключ для сравнения из функции, передаваемой методу. Вот две версии comparing():

```
static <T, U extends Comparable<? super U>> Comparator<T>
  comparing(Function<? super T, ? extends U> getKey)
static <T, U> Comparator<T>
  comparing(Function<? super T, ? extends U> getKey,
            Comparator<? super U> keyComp)
```

В обеих версиях getKey ссылается на функцию, которая получает следующий ключ для сравнения. Во второй версии в keyComp указывается компаратор, используемый для сравнения ключей. Кроме того, в интерфейс Comparator добавлены следующие специализированные версии этих методов для примитивных типов:

```
static <T> Comparator<T> comparingDouble(ToDoubleFunction<? super T> getKey)
static <T> Comparator<T> comparingInt(ToIntFunction<? super T> getKey)
static <T> Comparator<T> comparingLong(ToLongFunction<? super T> getKey)
```

Во всех методах getKey ссылается на функцию, получающую следующий ключ для сравнения.

Использование компаратора

Ниже приведен пример, демонстрирующий возможности специального компаратора. Он реализует метод compare() для строк, который работает противоположно нормальному компаратору. В результате древовидный набор сортируется в обратном порядке.

```
// Использование специального компаратора.
import java.util.*;

// Обратный компаратор для строк.
```

```
class MyComp implements Comparator<String> {
  public int compare(String aStr, String bStr) {
    // Обратить сравнение.
    return bStr.compareTo(aStr);
  }

  //Переопределять equals() и методы со стандартной реализацией не требуется
}

class CompDemo {
  public static void main(String[] args) {
    // Создать древовидный набор.
    TreeSet<String> ts = new TreeSet<String>(new MyComp());

    // Добавить элементы в древовидный набор.
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");

    // Отобразить элементы.
    for(String element : ts)
      System.out.print(element + " ");

    System.out.println();
  }
}
```

Как видно в выводе программы, древовидный набор теперь сортируется в обратном порядке:

```
F E D C B A
```

Внимательно взгляните на класс `MyComp`, который реализует интерфейс `Comparator` путем реализации метода `compare()`. (Как объяснялось ранее, переопределение метода `equals()` не является ни необходимым, ни общепринятым. Также нет нужды переопределять методы со стандартной реализацией.) Внутри `compare()` метод `compareTo()` класса `String` сравнивает две строки. Тем не менее, метод `compareTo()` вызывается на экземпляре `bStr`, а не на `aStr`, приводя к обратному результату сравнения.

Хотя способ реализации компаратора с обратным упорядочением в предыдущей программе вполне адекватен, решить задачу можно и по-другому. Теперь имеется возможность просто вызвать `reverse()` для компаратора с естественным упорядочением. Метод `reverse()` возвратит эквивалентный компаратор, но он будет работать в обратном порядке. Скажем, в предыдущей программе класс `MyComp` можно переписать в виде компаратора с естественным упорядочением:

```
class MyComp implements Comparator<String> {
  public int compare(String aStr, String bStr) {
    return aStr.compareTo(bStr);
  }
}
```

Затем с применением следующего фрагмента кода можно создать объект TreeSet, который располагает свои строковые элементы в обратном порядке:

```
MyComp mc = new MyComp(); // Создать компаратор.

// Передать конструктору TreeSet версию MyComp с обратным порядком.
TreeSet<String> ts = new TreeSet<String>(mc.reversed());
```

Если вы вставите показанный выше новый код в предыдущую программу, то результаты окажутся такими же, как и ранее. В этом случае использование reverse() не дает никаких преимуществ. Однако в тех случаях, когда требуется создать компаратор и с естественным, и с обратным порядком, метод reversed() будет простым способом получения компаратора с обратным порядком, не кодируя его явно.

На самом деле в предыдущих примерах нет необходимости создавать класс MyComp, поскольку вместо него допускается применять лямбда-выражение. Например, с использованием следующего оператора можно полностью избавиться от класса MyComp и создать компаратор строк:

```
// Использовать лямбда-выражение для реализации Comparator<String>.
Comparator<String> mc = (aStr, bStr) -> aStr.compareTo(bStr);
```

Еще один момент: в этом простом примере есть возможность указать обратный компаратор посредством лямбда-выражения прямо в вызове конструктора TreeSet():

```
//Передать обратный компаратор конструктору TreeSet() через лямбда-выражение
TreeSet<String> ts = new TreeSet<String>(
                     (aStr, bStr) -> bStr.compareTo(aStr));
```

Благодаря внесению таких изменений программа существенно укорачивается, как показано ниже в ее окончательной версии:

```
// Использовать лямбда-выражение для создания обратного компаратора.
import java.util.*;

class CompDemo2 {
  public static void main(String[] args) {

    // Передать обратный компаратор конструктору TreeSet()
    // через лямбда-выражение.
    TreeSet<String> ts = new TreeSet<String>(
                         (aStr, bStr) -> bStr.compareTo(aStr));
    // Добавить элементы в древовидный набор.
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");
    // Отобразить элементы.
    for(String element : ts)
      System.out.print(element + " ");
    System.out.println();
  }
}
```

Рассмотрим более реалистичный пример применения специального компаратора. Следующая программа представляет собой обновленную версию показанной ранее программы с сохранением балансов счетов в TreeMap. В предыдущей версии счета сортировались по имени и фамилии владельца, но сортировка начиналась с имени. Теперь счета будут сортироваться по фамилии, для чего используется компаратор, который сравнивает фамилию владельца каждого счета. В результате карта окажется отсортированной по фамилии.

```
// Использование компаратора для сортировки счетов по фамилии владельца.
import java.util.*;

// Сравнивает последние полные слова в двух строках.
class TComp implements Comparator<String> {
  public int compare(String aStr, String bStr) {
    int i, j, k;

    // Найти индекс, начинающийся с фамилии.
    i = aStr.lastIndexOf(' ');
    j = bStr.lastIndexOf(' ');

    k = aStr.substring(i).compareToIgnoreCase (bStr.substring(j));
    if(k==0) // фамилии совпадают, проверить полное имя
      return aStr.compareToIgnoreCase (bStr);
    else
      return k;
  }
  // Переопределять equals() не нужно.
}
class TreeMapDemo2 {
  public static void main(String[] args) {
    // Создать древовидную карту.
    TreeMap<String, Double> tm = new TreeMap<String, Double>(new TComp());

    // Поместить элементы в карту.
    tm.put("John Doe", 3434.34);
    tm.put("Tom Smith", 123.22);
    tm.put("Jane Baker", 1378.00);
    tm.put("Tod Hall", 99.22);
    tm.put("Ralph Smith", -19.08);

    // Получить набор элементов.
    Set<Map.Entry<String, Double>> set = tm.entrySet();

    // Отобразить элементы.
    for(Map.Entry<String, Double> me : set) {
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }
    System.out.println();
    // Пополнить счет клиента John Doe на 1000.
    double balance = tm.get("John Doe");
    tm.put("John Doe", balance + 1000);
    System.out.println("Новый баланс клиента John Doe: " + tm.get("John Doe"));
  }
}
```

Вот вывод; обратите внимание, что счета теперь отсортированы по фамилии:

```
Jane Baker: 1378.0
John Doe: 3434.34
Todd Hall: 99.22
Ralph Smith: -19.08
Tom Smith: 123.22

Новый баланс клиента John Doe: 4434.34
```

Класс компаратора TComp обеспечивает сравнение двух строк, содержащих имена и фамилии. Сначала сравниваются фамилии, для чего в каждой строке ищется индекс последнего пробела, а затем сравниваются подстроки каждого элемента, начинающиеся в этой точке. В случаях, когда фамилии эквивалентны, сравниваются имена. В итоге получается древовидная карта, отсортированная по фамилии и в пределах фамилии по имени. Результат легко заметить, потому что в выводе клиент Ralph Smith находится перед клиентом Tom Smith.

Предыдущую программу можно переписать так, чтобы карта сортировалась по фамилии и затем по имени. Такой подход предусматривает применение метода thenComparing(). Вспомните, что метод thenComparing() позволяет указать второй компаратор, который будет использоваться, если вызывающий компаратор возвращает признак равенства. Ниже показано, как выглядит переделанная программа:

```
// Использование thenComparing() для сортировки счетов по фамилии
// и затем по имени владельца.
import java.util.*;

// Компаратор, который сравнивает фамилии.
class CompLastNames implements Comparator<String> {
  public int compare(String aStr, String bStr) {
    int i, j;

    // Найти индекс начала фамилии.
    i = aStr.lastIndexOf(' ');
    j = bStr.lastIndexOf(' ');

    return aStr.substring(i).compareToIgnoreCase(bStr.substring(j));
  }
}

// Сортировать по полному имени, когда фамилии одинаковы.
class CompThenByFirstName implements Comparator<String> {
  public int compare(String aStr, String bStr) {
    int i, j;

    return aStr.compareToIgnoreCase(bStr);
  }
}

class TreeMapDemo2A {
  public static void main(String[] args) {
    // Использовать thenComparing() для создания компаратора, который
    // сравнивает фамилии и затем полные имена, когда фамилии совпадают.
```

```
    CompLastNames compLN = new CompLastNames();
    Comparator<String> compLastThenFirst =
      compLN.thenComparing(new CompThenByFirstName());

    // Создать древовидную карту.
    TreeMap<String, Double> tm =
      new TreeMap<String, Double>(compLastThenFirst);

    // Поместить элементы в карту.
    tm.put("John Doe", 3434.34);
    tm.put("Tom Smith", 123.22);
    tm.put("Jane Baker", 1378.00);
    tm.put("Tod Hall", 99.22);
    tm.put("Ralph Smith", -19.08);

    // Получить набор элементов.
    Set<Map.Entry<String, Double>> set = tm.entrySet();

    // Отобразить элементы.
    for(Map.Entry<String, Double> me : set) {
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }
    System.out.println();

    // Пополнить счет клиента John Doe на 1000.
    double balance = tm.get("John Doe");
    tm.put("John Doe", balance + 1000);

    System.out.println("Новый баланс клиента John Doe: " + tm.get("John Doe"));
  }
}
```

Новая версия программы производит тот же вывод, что и предыдущая. Она отличается только способом выполнения своей работы. Для начала обратите внимание на создание компаратора по имени `CompLastNames`, сравнивающего только фамилии. Второй компаратор, `CompThenByFirstName`, сравнивает полное имя, которое включает имя и фамилию, начиная с имени. Далее создается объект `TreeMap`:

```
CompLastNames compLN = new CompLastNames();
Comparator<String> compLastThenFirst =
  compLN.thenComparing(new CompThenByFirstName());
```

Основным компаратором является `compLN`, который представляет собой экземпляр `CompLastNames`. Для него вызывается метод `thenComparing()` с передачей экземпляра `CompThenByFirstName`. Результат присваивается компаратору по имени `compLastThenFirst`, который применяется для конструирования объекта `TreeMap`:

```
TreeMap<String, Double> tm =
  new TreeMap<String, Double>(compLastThenFirst);
```

Теперь всякий раз, когда фамилии сравниваемых элементов одинаковы, для их упорядочивания используется полное имя, начиная с просто имени, т.е. имена упорядочиваются по фамилии, а внутри фамилий — по имени.

И последнее замечание: ради большей ясности в рассмотренном примере явно создаются два класса компараторов с именами CompLastNames и ThenByFirstNames, но взамен можно было бы применять лямбда-выражения. Попробуйте сделать это самостоятельно. Просто следуйте тому же общему подходу, который описан для показанного ранее примера CompDemo2.

Алгоритмы коллекций

В инфраструктуре Collections Framework определен ряд алгоритмов, которые можно применять к коллекциям и картам. Такие алгоритмы определены в виде статических методов класса Collections и кратко описаны в табл. 20.19.

Таблица 20.19. Алгоритмы, определенные в классе Collections

Метод	Описание
`static <T> boolean addAll(Collection <? super T> c, T ... elements)`	Вставляет элементы, переданные в elements, в коллекцию, указанную в c. Возвращает true, если элементы были добавлены, или false в противном случае
`static <T> Queue<T> asLifoQueue(Deque<T> c)`	Возвращает представление коллекции c, работающее по принципу "последним пришел — первым обслужен"
`static <T> int binarySearch (List<? extends T> list, T value, Comparator <? super T> c)`	Ищет значение value в списке list, упорядоченном согласно компаратору c. Возвращает позицию value в list или отрицательное значение, если value не найдено
`static <T> int binarySearch (List<? extends Comparable<? super T>> list, T value)`	Ищет значение value в списке list. Список должен быть отсортирован. Возвращает позицию value в list или отрицательное значение, если value не найдено
`static <E> Collection<E> checkedCollection (Collection<E> c, Class<E> t)`	Возвращает безопасное к типам представление времени выполнения для объекта Collection. Попытка вставки несовместимого элемента приведет к генерации исключения ClassCastException
`static <E> List<E> checkedList(List<E> c, Class<E> t)`	Возвращает безопасное к типам представление времени выполнения для объекта List. Попытка вставки несовместимого элемента приведет к генерации исключения ClassCastException

Продолжение табл. 20.19

Метод	Описание
`static <K, V> Map<K, V> checkedMap(Map<K, V> c, Class<K> keyT, Class<V> valueT)`	Возвращает безопасное к типам представление времени выполнения для объекта Map. Попытка вставки несовместимого элемента приведет к генерации исключения ClassCastException
`static <K,V> NavigableMap <K,V> checkedNavigableMap (NavigableMap<K, V> nm, Class<E> keyT, Class<V> valueT)`	Возвращает безопасное к типам представление времени выполнения для объекта NavigableMap. Попытка вставки несовместимого элемента приведет к генерации исключения ClassCastException
`static <E> NavigableSet<E> checkedNavigableSet (NavigableSet<E> ns, Class<E> t)`	Возвращает безопасное к типам представление времени выполнения для объекта NavigableSet. Попытка вставки несовместимого элемента приведет к генерации исключения ClassCastException
`static <E> Queue<E> checkedQueue(Queue<E> q, Class<E> t)`	Возвращает безопасное к типам представление времени выполнения для объекта Queue. Попытка вставки несовместимого элемента приведет к генерации исключения ClassCastException
`static <E> List<E> checkedSet(Set<E> c, Class<E> t)`	Возвращает безопасное к типам представление времени выполнения для объекта Set. Попытка вставки несовместимого элемента приведет к генерации исключения ClassCastException
`static <K, V> SortedMap<K, V> checkedSortedMap(SortedMap <K, V> c, Class<K> keyT, Class<V> valueT)`	Возвращает безопасное к типам представление времени выполнения для объекта SortedMap. Попытка вставки несовместимого элемента приведет к генерации исключения ClassCastException
`static <E> SortedSet<E> checkedSortedSet (SortedSet<E> c, Class<E> t)`	Возвращает безопасное к типам представление времени выполнения для объекта SortedSet. Попытка вставки несовместимого элемента приведет к генерации исключения ClassCastException
`static <T> void copy (List<? super T> list1, List<? extends T> list2)`	Копирует элементы из list2 в list1

Продолжение табл. 20.19

Метод	Описание
`static boolean disjoint(Collection<?> a, Collection<?> b)`	Сравнивает элементы в a с элементами в b. Возвращает `true`, если две коллекции не содержат общих элементов (т.е. в коллекциях находятся непересекающиеся множества элементов), или `false` в противном случае
`static <T> Enumeration<T> emptyEnumeration()`	Возвращает пустое перечисление, т.е. не содержащее никаких элементов
`static <T> Iterator<T> emptyIterator()`	Возвращает пустой итератор, т.е. не содержащий никаких элементов
`static <T> List<T> emptyList()`	Возвращает неизменяемый пустой объект `List` выведенного типа
`static <T> ListIterator<T> emptyListIterator()`	Возвращает пустой списковый итератор, т.е. не содержащий никаких элементов
`static <K, V> Map<K, V> emptyMap()`	Возвращает неизменяемый пустой объект `Map` выведенного типа
`static <K, V> NavigableMap<K, V> emptyNavigableMap()`	Возвращает неизменяемый пустой объект `NavigableMap` выведенного типа
`static <E> NavigableSet<E> emptyNavigableSet()`	Возвращает неизменяемый пустой объект `NavigableSet` выведенного типа
`static <T> Set<T> emptySet()`	Возвращает неизменяемый пустой объект `Set` выведенного типа
`static <K, V> SortedMap<K, V> emptySortedMap()`	Возвращает неизменяемый пустой объект `SortedMap` выведенного типа
`static <E> SortedSet<E> emptySortedSet()`	Возвращает неизменяемый пустой объект `SortedSet` выведенного типа
`static <T> Enumeration<T> enumeration(Collection<T> c)`	Возвращает перечисление по коллекции c. (См. раздел "Интерфейс `Enumeration`" далее в главе.)
`static <T> void fill (List<? super T> list, T obj)`	Присваивает объект `obj` каждому элементу в списке `list`
`static int frequency(Collection<?> c, object obj)`	Подсчитывает количество вхождений объекта `obj` в коллекции c и возвращает результат

Продолжение табл. 20.19

Метод	Описание
`static int indexOfSubList(List<?> list, List<?> subList)`	Ищет в списке `list` первое вхождение `subList`. Возвращает индекс первого совпадения или `-1`, если оно не найдено
`static int lastIndexOfSubList(List<?> list, List<?> subList)`	Ищет в списке `list` последнее вхождение `subList`. Возвращает индекс последнего совпадения или `-1`, если оно не найдено
`static <T> ArrayList<T> list(Enumeration<T> enum)`	Возвращает объект `ArrayList`, который содержит элементы `enum`
`static <T> T max (Collection<? extends T> c, Comparator<? super T> comp)`	Возвращает наибольший элемент в коллекции `c`, как определено компаратором `comp`
`static <T extends Object & Comparable<? super T>> T max(Collection <? extends T> c)`	Возвращает наибольший элемент в коллекции `c`, как определено естественным упорядочением. Сортировать коллекцию не нужно
`static <T> T min (Collection<? extends T> c, Comparator<? super T> comp)`	Возвращает наименьший элемент в коллекции `c`, как определено компаратором `comp`. Сортировать коллекцию не нужно
`static <T extends Object & Comparable<? super T>>T min (Collection<? extends T> c)`	Возвращает наименьший элемент в коллекции `c`, как определено естественным упорядочением
`static <T> List<T> nCopies(int num, T obj)`	Возвращает `num` копий объекта `obj`, содержащихся в неизменяемом списке. Значение `num` должно быть больше или равно нулю
`static <E> Set<E> newSetFromMap(Map<E, Boolean> m)`	Создает и возвращает набор, поддерживаемый в актуальном состоянии указанной в `m` картой, которая на момент вызова этого метода должна быть пустой
`static <T> boolean replaceAll (List<T> list, T old, T new)`	Заменяет в списке `list` все вхождения объекта `old` объектом `new`. Возвращает `true` при наличии хотя бы одной замены или `false` в противном случае
`static void reverse (List<T> list)`	Изменяет порядок следования элементов в списке `list` на противоположный

Продолжение табл. 20.19

Метод	Описание
`static <T> Comparator<T> reverseOrder (Comparator<T> comp)`	Возвращает обратный компаратор, основанный на компараторе, который передан в comp. То есть возвращаемый компаратор изменяет на противоположный результат сравнения, в котором используется comp
`static <T> Comparator<T> reverseOrder()`	Возвращает обратный компаратор, который изменяет на противоположный результат сравнения двух элементов
`static void rotate (List<T> list, int n)`	Сдвигает список list на n позиций вправо. Для сдвига влево понадобится передать в n отрицательное значение
`static void shuffle (List<T> list, Random r)`	Тасует (т.е. рандомизирует) элементы в списке list, используя r в качестве источника случайных чисел
`static void shuffle(List<T> list)`	Тасует (т.е. рандомизирует) элементы в списке list
`static <T> Set<T> singleton(T obj)`	Возвращает объект obj в виде неизменяемого набора. Предоставляет легкий способ преобразования одиночного объекта в набор
`static <T> List<T> singletonList(T obj)`	Возвращает объект obj в виде неизменяемого списка. Предоставляет легкий способ преобразования одиночного объекта в список
`static <K, V> Map<K, V> singletonMap(K k, V v)`	Возвращает пару "ключ-значение" k, v в виде неизменяемой карты. Предоставляет легкий способ преобразования одиночной пары "ключ-значение" в карту
`static <T> void sort(List<T> list, Comparator<? super T> comp)`	Сортирует элементы списка list в соответствии с компаратором comp
`static <T extends Comparable<? super T>> void sort(List<T> list)`	Сортирует элементы списка list в соответствии с натуральным упорядочением
`static void swap(List<?> list, int idx1, int idx2)`	Меняет местами элементы в списке list по индексам, указанным в idx1 и idx2

Продолжение табл. 20.19

Метод	Описание
`static <T> Collection<T> synchronizedCollection (Collection<T> c)`	Возвращает безопасную к потокам коллекцию, поддерживаемую в актуальном состоянии посредством c
`static <T> List<T> synchronizedList (List<T> list)`	Возвращает безопасный к потокам список, поддерживаемый в актуальном состоянии посредством list
`static <K, V> Map<K, V> synchronizedMap (Map<K, V> m)`	Возвращает безопасную к потокам карту, поддерживаемую в актуальном состоянии посредством m
`static <K, V> NavigableMap<K, V> synchronizedNavigableMap (NavigableMap<K, V> nm)`	Возвращает синхронизированную навигационную карту, поддерживаемую в актуальном состоянии посредством nm
`static <T> NavigableSet<T> synchronizedNavigableSet (NavigableSet<T> ns)`	Возвращает синхронизированный навигационный набор, поддерживаемый в актуальном состоянии посредством ns
`static <T> Set<T> synchronizedSet(Set<T> s)`	Возвращает безопасный к потокам набор, поддерживаемый в актуальном состоянии посредством s
`static <K, V> SortedMap <K,V>synchronizedSortedMap (SortedMap<K, V> sm)`	Возвращает безопасную к потокам отсортированную карту, поддерживаемую в актуальном состоянии посредством sm
`static <T> SortedSet<T> synchronizedSortedSet (SortedSet<T> ss)`	Возвращает безопасный к потокам отсортированный набор, поддерживаемый в актуальном состоянии посредством ss
`static <T> Collection<T> unmodifiableCollection (Collection<? extends T> c)`	Возвращает неизменяемую коллекцию, поддерживаемую в актуальном состоянии посредством c
`static <T> List<T> unmodifiableList (List<? extends T> list)`	Возвращает неизменяемый список, поддерживаемый в актуальном состоянии посредством list
`static <K, V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m)`	Возвращает неизменяемую карту, поддерживаемую в актуальном состоянии посредством m
`static<K,V>NavigableMap<K,V> unmodifiableNavigableMap (NavigableMap<K, ? extends V> nm)`	Возвращает неизменяемую навигационную карту, поддерживаемую в актуальном состоянии посредством nm

Окончание табл. 20.19

Метод	Описание
`static <T> NavigableSet<T> unmodifiableNavigableSet (NavigableSet<T> ns)`	Возвращает неизменяемый навигационный набор, поддерживаемый в актуальном состоянии посредством ns
`static <T> Set<T> unmodifiableSet (Set<? extends T> s)`	Возвращает неизменяемый набор, поддерживаемый в актуальном состоянии посредством s
`static <K,V> ortedMap<K,V> unmodifiableSortedMap(Sort edMap<K, ? extends V> sm)`	Возвращает неизменяемую отсортированную карту, поддерживаемую в актуальном состоянии посредством sm
`static <T> SortedSet<T> unmodifiableSortedSet (SortedSet<T> ss)`	Возвращает неизменяемый отсортированный набор, поддерживаемый в актуальном состоянии посредством ss

Некоторые методы могут генерировать исключение `ClassCastException` при попытке сравнения несовместимых типов либо исключение `UnsupportedOperationException` при попытке модификации неизменяемой коллекции. В зависимости от метода возможны и другие исключения.

Особое внимание заслуживает набор методов с именами, начинающимися на checked, вроде метода `checkedCollection()`, который возвращает то, что в документации по API называется представлением коллекции, динамически безопасным в отношении типов. Такое представление является ссылкой на коллекцию, которая отслеживает операции вставки в коллекцию на предмет совместимости типов во время выполнения. Попытка вставки несовместимого элемента приводит к генерации исключения `ClassCastException`. Использовать представление подобного рода особенно полезно во время отладки, поскольку оно гарантирует, что коллекция всегда содержит допустимые элементы. В числе связанных методов входят `checkedSet()`, `checkedList()`, `checkedMap()` и т.д. Они получают безопасное к типам представление для указанной коллекции.

Обратите внимание, что несколько методов, например, `synchronizedList()` и `synchronizedSet()`, служат для получения синхронизированных (*безопасных к потокам*) копий разнообразных коллекций. Как правило, стандартные реализации коллекций не являются синхронизированными. Для обеспечения синхронизации потребуется применять алгоритмы синхронизации. Еще один момент: итераторы для синхронизированных коллекций должны использоваться внутри блоков `synchronized`.

Набор методов с именами, начинающимися с `unmodifiable`, возвращает представления различных коллекций, которые нельзя модифицировать. Они полезны, когда какому-то процессу необходимо предоставить возможность чтения, но не записи в коллекцию.

В классе Collections определены три статические переменные: EMPTY_SET, EMPTY_LIST и EMPTY_MAP. Все они неизменяемы.

В показанной ниже программе демонстрируется работа некоторых алгоритмов. В ней создается и инициализируется связный список. Метод reverseOrder() возвращает компаратор, который изменяет на противоположный результат сравнения объектов Integer. Элементы списка сортируются в соответствии с этим компаратором и затем отображаются. Далее список рандомизируется вызовом shuffle(), после чего отображаются его наименьшее и наибольшее значения.

```
// Демонстрация работы разнообразных алгоритмов.
import java.util.*;

class AlgorithmsDemo {
  public static void main(String[] args) {

    // Создать и инициализировать связный список.
    LinkedList<Integer> ll = new LinkedList<Integer>();
    ll.add(-8);
    ll.add(20);
    ll.add(-20);
    ll.add(8);

    // Создать компаратор с обратным порядком.
    Comparator<Integer> r = Collections.reverseOrder();

    // Сортировать список с использованием созданного компаратора.
    Collections.sort(ll, r);

    System.out.print("Список отсортирован в обратном порядке: ");
    for(int i : ll)
      System.out.print(i+ " ");

    System.out.println();

    // Тасовать список.
    Collections.shuffle(ll);

    // Отобразить рандомизированный список.
    System.out.print("Список перетасован: ");
    for(int i : ll)
      System.out.print(i + " ");

    System.out.println();

    System.out.println("Наименьшее значение: " + Collections.min(ll));
    System.out.println("Наибольшее значение: " + Collections.max(ll));
  }
}
```

Вот вывод программы:

```
Список отсортирован в обратном порядке: 20 8 -8 -20
Список перетасован: 20 -20 8 -8
Наименьшее значение: -20
Наибольшее значение: 20
```

Обратите внимание, что методы min() и max() оперируют со списком после его тасования. Ни один из них не требует отсортированного списка для своей работы.

Массивы

Класс Arrays предоставляет различные статические служебные методы, полезные при работе с массивами. Эти методы помогают преодолеть брешь между коллекциями и массивами. В данном разделе рассматриваются все методы, определенные в Arrays.

Метод asList() возвращает список, поддерживаемый указанным массивом. Другими словами, и список, и массив ссылаются на одно и то же местоположение в памяти. Метод asList() имеет следующую сигнатуру:

```
static <T> List asList(T... array)
```

Здесь array — это массив, который содержит данные.

Для нахождения указанного значения метод binarySearch() задействует двоичный поиск. Этот метод необходимо применять к отсортированным массивам. Ниже перечислены некоторые его формы (дополнительные формы позволяют выполнять поиск в поддиапазоне):

```
static int binarySearch(byte[] array, byte value)
static int binarySearch(char[] array, char value)
static int binarySearch(double[] array, double value)
static int binarySearch(float[] array, float value)
static int binarySearch(int[] array, int value)
static int binarySearch(long[] array, long value)
static int binarySearch(short[] array, short value)
static int binarySearch(Object[] array, Object value)
static <T> int binarySearch(T[] array, T value, Comparator<? super T> c)
```

Здесь array — массив для поиска, а value — значение, которое нужно найти. Последние две формы генерируют исключение ClassCastException, если массив содержит элементы, которые сравнить невозможно (например, Double и StringBuffer), или если значение несовместимо с типами в массиве. В последней форме для определения порядка элементов в массиве используется компаратор c. Во всех случаях, если значение существует в массиве, тогда возвращается индекс элемента, а в противном случае — отрицательное значение.

Метод copyOf() возвращает копию массива и имеет следующие формы:

```
static boolean[] copyOf(boolean[] source, int len)
static byte[] copyOf(byte[] source, int len)
static char[] copyOf(char[] source, int len)
static double[] copyOf(double[] source, int len)
static float[] copyOf(float[] source, int len)
static int[] copyOf(int[] source, int len)
static long[] copyOf(long[] source, int len)
static short[] copyOf(short[] source, int len)
static <T> T[] copyOf(T[] source, int len)
static <T,U> T[] copyOf(U[] source, int len, Class<? extends T[]> resultT)
```

Исходный массив указывается в source, а длина копии — в len. Если копия длиннее, чем source, тогда копия дополняется нулями (для числовых массивов), null (для объектных массивов) или false (для булевских массивов). Если копия короче, чем source, то копия усекается. В последней форме тип resultT становится типом возвращаемого массива. Если len имеет отрицательное значение, тогда генерируется исключение NegativeArraySizeException. Если источник имеет значение null, то возникает исключение NullPointerException. Если resultT не является совместимым с типом source, тогда генерируется исключение ArrayStoreException. Метод copyOfRange() возвращает копию диапазона внутри массива и имеет перечисленные далее формы:

```
static boolean[] copyOfRange(boolean[] source, int start, int end)
static byte[] copyOfRange(byte[] source, int start, int end)
static char[] copyOfRange(char[] source, int start, int end)
static double[] copyOfRange(double[] source, int start, int end)
static float[] copyOfRange(float[] source, int start, int end)
static int[] copyOfRange(int[] source, int start, int end)
static long[] copyOfRange(long[] source, int start, int end)
static short[] copyOfRange(short[] source, int start, int end)
static <T> T[] copyOfRange(T[] source, int start, int end)
static <T,U> T[] copyOfRange(U[] source, int start, int end,
  Class<? extends T[]> resultT)
```

Исходный массив указывается в source. Диапазон для копирования определяется индексами, передаваемыми через start и end. Диапазон простирается от start до end-1. Если диапазон длиннее, чем source, то копия дополняется нулями (для числовых массивов), null (для объектных массивов) или false (для булевских массивов). В последней форме тип resultT становится типом возвращаемого массива. Если start имеет отрицательное значение или превышает длину source, тогда генерируется исключение ArrayIndexOutOfBoundsException. Если start больше end, то возникает исключение IllegalArgumentException. Если source имеет значение null, тогда генерируется исключение NullPointerException. Если resultT не является совместимым с типом source, то генерируется исключение ArrayStoreException.

Метод equals() возвращает true, если два массива эквивалентны. В противном случае возвращается false. Ниже показано несколько его форм. Вдобавок доступны версии, которые позволяют указывать диапазон, обобщенный тип массива и/или компаратор.

```
static boolean equals(boolean[] array1, boolean[] array2)
static boolean equals(byte[] array1, byte[] array2)
static boolean equals(char[] array1, char[] array2)
static boolean equals(double[] array1, double[] array2)
static boolean equals(float[] array1, float[] array2)
static boolean equals(int[] array1, int[] array2)
static boolean equals(long[] array1, long[] array2)
static boolean equals(short[] array1, short[] array2)
static boolean equals(Object[] array1, Object[] array2)
```

В приведенных формах array1 и array2 — два массива, которые сравниваются на предмет равенства.

Метод deepEquals() допускается применять для определения равенства двух массивов, которые могут включать вложенные массивы. Вот его объявление:

```
static boolean deepEquals(Object[] a, Object[] b)
```

Он возвращает true, если массивы, переданные в a и b, содержат те же самые элементы. Если a и b включают вложенные массивы, то содержимое этих вложенных массивов тоже проверяется. Метод возвращает false, если массивы или любые вложенные массивы различаются.

Метод fill() присваивает значение всем элементам массива. Другими словами, он заполняет массив указанным значением. Метод fill() имеет две разновидности. Первая разновидность со следующими формами заполняет целый массив:

```
static void fill(boolean[] array, boolean value)
static void fill(byte[] array, byte value)
static void fill(char[] array, char value)
static void fill(double[] array, double value)
static void fill(float[] array, float value)
static void fill(int[] array, int value)
static void fill(long[] array, long value)
static void fill(short[] array, short value)
static void fill(Object[] array, Object value)
```

Всем элементам массива array присваивается значение value. Вторая разновидность метода fill() присваивает значение подмножеству массива.

Метод sort() сортирует массив в возрастающем порядке и имеет две версии. Первая разновидность сортирует весь массив:

```
static void sort(byte[] array)
static void sort(char[] array)
static void sort(double[] array)
static void sort(float[] array)
static void sort(int[] array)
static void sort(long[] array)
static void sort(short[] array)
static void sort(Object[] array)
static <T> void sort(T[] array, Comparator<? super T> c)
```

В array указывается массив, подлежащий сортировке. В последней форме в c передается компаратор, который используется для упорядочения элементов массива. Последние две формы могут генерировать исключение ClassCastException, если элементы сортируемого массива несопоставимы. Вторая разновидность sort() позволяет указывать диапазон внутри массива, который нужно отсортировать.

Одним из довольно мощных методов класса Arrays является parallelSort(), поскольку он сортирует в возрастающем порядке части массива параллельно и затем объединяет результаты. Такой подход может

значительно сократить время сортировки. Подобно `sort()` существуют две базовых разновидности метода `parallelSort()`, для каждой из которых определено несколько перегруженных версий. Первая разновидность сортирует весь массив:

```
static void parallelSort(byte[] array)
static void parallelSort(char[] array)
static void parallelSort(double[] array)
static void parallelSort(float[] array)
static void parallelSort(int[] array)
static void parallelSort(long[] array)
static void parallelSort(short[] array)
static <T extends Comparable<? super T>> void parallelSort(T[] array)
static <T> void parallelSort(T[] array, Comparator<? super T> c)
```

В `array` указывается массив, подлежащий сортировке. В последней форме в `c` передается компаратор, который применяется для упорядочения элементов массива. Последние две формы могут генерировать исключение `ClassCastException`, если элементы сортируемого массива несопоставимы. Вторая разновидность `parallelSort()` позволяет указывать диапазон внутри массива, который нужно отсортировать.

Класс `Arrays` поддерживает сплитераторы за счет включения метода `spliterator()`, который имеет две основные разновидности. Первая разновидность возвращает разделитель для всего массива:

```
static Spliterator.OfDouble spliterator(double[] array)
static Spliterator.OfInt spliterator(int[] array)
static Spliterator.OfLong spliterator(long[] array)
static <T> Spliterator spliterator(T[] array)
```

Здесь `array` — это массив, через который будет проходить сплитератор. Вторая разновидность метода `spliterator()` позволяет указывать диапазон для итерации внутри массива.

Класс `Arrays` поддерживает интерфейс `Stream` путем реализации метода `stream()`, имеющего две разновидности. Вот первая:

```
static DoubleStream stream(double[] array)
static IntStream stream(int[] array)
static LongStream stream(long[] array)
static <T> Stream stream(T[] array)
```

В `array` передается массив, на который будет ссылаться поток. Вторая разновидность метода `stream()` позволяет указывать диапазон внутри массива.

Есть еще два связанных друг с другом метода: `setAll()` и `parallelSetAll()`. Оба присваивают значения всем элементам, но `parallelSetAll()` работает параллельно:

```
static void setAll(double[] array, IntToDoubleFunction<? extends T> genVal)
static void parallelSetAll(double[] array,
                           IntToDoubleFunction<? extends T> genVal)
```

Для каждого из них существует несколько перегруженных версий, которые обрабатывают типы `int`, `long` и обобщенные типы.

Один из наиболее интригующих методов, определенных в Arrays, называется parallelPrefix(); он изменяет массив таким образом, что каждый элемент содержит совокупный результат операции, примененной ко всем предыдущим элементам. Например, если речь идет об операции умножения, то элементы возвращенного массива будут содержать значения, ассоциированные с текущим произведением исходных значений. Для метода parallelPrefix() имеется несколько перегруженных версий. Вот один пример:

```
static void parallelPrefix(double[] array, DoubleBinaryOperator func)
```

В array указывается массив, над которым выполняется действие, а в func — применяемая операция. (DoubleBinaryOperator — это функциональный интерфейс, определенный в пакете java.util.function.) Предлагается множество других версий, в том числе версии для типов int, long и обобщенных типов, а также версии, позволяющие указывать диапазон внутри массива, с которым нужно работать.

В JDK 9 класс Arrays получил три метода сравнения: compare(), compareUnsigned() и mismatch(). Для каждого метода предусмотрено несколько перегруженных версий, и некоторые из них позволяют указывать диапазон для сравнения. Далее приводятся краткие описания всех упомянутых методов. Метод compare() сравнивает два массива. Он возвращает ноль, если они одинаковы, положительное значение, если первый массив больше второго, и отрицательное значение, если первый массив меньше второго. Чтобы выполнить беззнаковое сравнение двух массивов, содержащих целочисленные значения, необходимо использовать метод compareUnsigned(). Найти местоположение первого несовпадения между двумя массивами поможет метод mismatch(), который возвращает индекс несовпадения или -1, если массивы эквивалентны.

Класс Arrays также предлагает методы toString() и hashCode() для различных типов массивов. Кроме того, предусмотрены методы deepToString() и deepHashCode(), которые эффективно работают с массивами, содержащими вложенные массивы.

В следующей программе демонстрируется применение ряда методов класса Arrays:

```
// Демонстрация использования класса Arrays.import java.util.*;

class ArraysDemo {
  public static void main(String[] args) {

    // Разместить в памяти и инициализировать массив.
    int[] array = new int[10];
    for(int i = 0; i < 10; i++)
      array[i] = -3 * i;

    // Отобразить, отсортировать и снова отобразить содержимое массива.
    System.out.print("Исходное содержимое: ");
    display(array);
    Arrays.sort(array);
    System.out.print("Содержимое после сортировки: ");
    display(array);
```

```
    // Заполнить массив и отобразить его содержимое.
    Arrays.fill(array, 2, 6, -1);
    System.out.print("Содержимое после вызова fill(): ");
    display(array);
    // Отсортировать массив и отобразить его содержимое.
    Arrays.sort(array);
    System.out.print("Содержимое после повторной сортировки: ");
    display(array);
    // Двоичный поиск значения -9.
    System.out.print("Значение -9 находится в позиции ");
    int index = Arrays.binarySearch(array, -9);
    System.out.println(index);
  }
  static void display(int[] array) {
    for(int i: array)
      System.out.print(i + " ");
    System.out.println();
  }
}
```

Ниже показан вывод, генерируемый программой:

```
Исходное содержимое: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Содержимое после сортировки: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
Содержимое после вызова fill(): -27 -24 -1 -1 -1 -1 -9 -6 -3 0
Содержимое после повторной сортировки: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
Значение -9 находится в позиции 2
```

Унаследованные классы и интерфейсы

Как объяснялось в начале главы, ранние версии пакета `java.util` не включали инфраструктуру Collections Framework. Взамен в нем было определено несколько классов и интерфейс, предоставляющий специальный способ хранения объектов. После добавления коллекций (в версии J2SE 1.2) некоторые первоначальные классы были переделаны для поддержки интерфейсов коллекций и теперь формально являются частью Collections Framework. Тем не менее, когда функциональность унаследованного класса дублируется в современной коллекции, как правило, имеет смысл отдавать предпочтение более нового классу коллекции.

И еще один момент: ни один из современных классов коллекций, описанных в настоящей главе, не является синхронизированным, но все унаследованные классы синхронизированы. В некоторых ситуациях это различие может быть важным. Конечно, коллекции можно легко синхронизировать с использованием одного из алгоритмов из класса `Collections`. Ниже перечислены унаследованные классы, определенные в пакете `java.util`:

`Dictionary` `Hashtable` `Properties` `Stack` `Vector`

Существует один унаследованный интерфейс по имени `Enumeration`. В последующих разделах по очереди рассматриваются `Enumeration` и все унаследованные классы.

Интерфейс Enumeration

В интерфейсе Enumeration определены методы, с помощью которых можно организовать перечисление (получение по одному за раз) элементов в коллекции объектов. Этот унаследованный интерфейс был заменен интерфейсом Iterator. Хотя Enumeration не объявлен нерекомендуемым, он считается устаревшим для нового кода. Однако Enumeration потребляется несколькими методами, которые определены в унаследованных классах (вроде Vector и Properties), а также рядом других классов API. В версии JDK 5 интерфейс Enumeration был модернизирован с учетом обобщений. Вот его объявление:

```
interface Enumeration<E>
```

В E указывается тип перечисляемых элементов.

В Enumeration определены два абстрактных метода:

```
boolean hasMoreElements()
E nextElement()
```

В случае реализации метод hasMoreElements() должен возвращать true, пока еще есть элементы для извлечения, и false, если перечислению были подвергнуты все элементы. Метод nextElement() возвращает следующий объект в перечислении, т.е. при каждом вызове nextElement() из перечисления будет получен очередной объект. Он генерирует исключение NoSuchElementException, когда перечисление завершено.

В версии JDK 9 интерфейс Enumeration пополнился методом со стандартной реализацией по имени asIterator():

```
default Iterator<E> asIterator()
```

Метод asIterator() возвращает итератор для элементов перечисления. В результате он обеспечивает простой способ преобразования реализации Enumeration старого стиля в современную реализацию Iterator. Более того, если часть элементов в перечислении уже была прочитана до вызова asIterator(), то возвращаемый итератор будет обращаться только к оставшимся элементам.

Класс Vector

Класс Vector реализует динамический массив. Он похож на ArrayList, но с двумя отличиями: Vector является синхронизированным и содержит много унаследованных методов, которые дублируют функциональность методов, определенных в Collections Framework. С появлением коллекций класс Vector был переделан с целью расширения AbstractList и реализации интерфейса List. В выпуске JDK 5 он был модернизирован с учетом обобщений и переработан для реализации Iterable. Таким образом, класс Vector полностью совместим с коллекциями, а по его содержимому можно проходить с помощью расширенного цикла for.

Класс Vector объявлен следующим образом:

```
class Vector<E>
```

В E указывается тип элементов, которые будут сохранены.

А вот конструкторы класса Vector:

```
Vector()
Vector(int size)
Vector(int size, int incr)
Vector(Collection<? extends E> c)
```

Первая форма конструктора создает стандартный вектор с начальным размером 10 элементов. Вторая форма конструирует вектор, начальная емкость которого указана в size. Третья форма создает вектор, начальная емкость которого задается size, а приращение указывается в incr. Приращение устанавливает количество элементов, под которые будет выделяться память каждый раз, когда размер вектора увеличивается. Четвертая форма конструирует вектор, содержащий элементы коллекции c.

Все векторы получают какую-то начальную емкость. После исчерпания начальной емкости следующая попытка сохранения объекта в векторе приводит к тому, что экземпляр Vector автоматически выделяет место для этого объекта плюс добавочное пространство под дополнительные объекты. Выделяя больше памяти, чем требуется, экземпляр Vector уменьшает количество выделений памяти, которые должны выполняться по мере роста вектора. Такое сокращение важно из-за того, что операции выделения памяти сопряжены с затратами в плане времени. Объем добавочного пространства, выделяемого при каждом перераспределении, определяется приращением, которое было указано при создании вектора. Если приращение не задавалось, тогда размер вектора будет удваиваться с каждым циклом распределения.

В классе Vector определены следующие защищенные данные-члены:

```
int capacityIncrement;
int elementCount;
Object[] elementData;
```

Значение приращения хранится в capacityIncrement, Количество элементов, в текущий момент находящихся в векторе, хранится в elementCount. Массив, содержащий вектор, хранится в elementData.

Помимо методов коллекций из интерфейса List в классе Vector определено несколько унаследованных методов, которые кратко описаны в табл.20.20.

Таблица 20.20. Унаследованные методы, определенные в классе Vector

Метод	Описание
`void addElement (E element)`	Добавляет в вектор объект, указанный в element
`int capacity()`	Возвращает емкость вектора
`Object clone()`	Возвращает дубликат вызывающего вектора
`boolean contains (Object element)`	Возвращает true, если элемент, указанный в element, содержится в векторе, или false в противном случае

Продолжение табл. 20.20

Метод	Описание
`void copyInto (Object[] array)`	Копирует элементы, содержащиеся в вызывающем векторе, в массив, указанный в `array`
`E elementAt (int index)`	Возвращает элемент в позиции, указанной с помощью `index`
`Enumeration<E> elements()`	Возвращает перечисление элементов в векторе
`void ensureCapacity (int size)`	Устанавливает минимальную емкость вектора в `size`
`E firstElement()`	Возвращает первый элемент в векторе
`int indexOf (Object element)`	Возвращает индекс первого вхождения элемента, указанного в `element`. Если элемент в векторе отсутствует, тогда возвращается `-1`
`int indexOf(Object element, int start)`	Возвращает индекс первого вхождения элемента, указанного в `element`, начиная с позиции `start`. Если элемент в векторе отсутствует, тогда возвращается `-1`
`void insertElementAt (Eelement,intindex)`	Добавляет элемент `element` в вектор в позицию, указанную с помощью `index`
`boolean isEmpty()`	Возвращает `true`, если вектор пуст, или `false`, если он содержит один и более элементов
`E lastElement()`	Возвращает последний элемент в векторе
`int lastIndexOf (Object element)`	Возвращает индекс последнего вхождения элемента, указанного в `element`. Если элемент в векторе отсутствует, тогда возвращается `-1`
`int lastIndexOf (Object element, int start)`	Возвращает индекс последнего вхождения элемента, указанного в `element`, перед позицией `start`. Если элемент в векторе отсутствует, тогда возвращается `-1`
`void removeAllElements()`	Опустошает вектор. После выполнения этого метода размер вектора становится нулевым
`boolean removeElement (Object element)`	Удаляет из вектора элемент, указанный в `element`. Если в векторе присутствует более одного экземпляра `element`, тогда удаляется первый. Возвращает `true` в случае успеха и `false`, если элемент не найден
`void removeElementAt (int index)`	Удаляет из вектора элемент в позиции, указанной в `index`

Окончание табл. 20.20

Метод	Описание
`void setElementAt (E element, int index)`	Назначает элементу, указанному в `element`, позицию, заданную в `index`
`void setSize(int size)`	Устанавливает количество элементов внутри вектора в `size`. Если новый размер меньше старого, то элементы утрачиваются. Если новый размер меньше старого, тогда добавляются элементы `null`
`int size()`	Возвращает количество элементов, содержащихся в векторе в текущий момент
`String toString()`	Возвращает строковый эквивалент вектора
`void trimToSize()`	Устанавливает емкость вектора равной количеству элементов, содержащихся в векторе в текущий момент

Поскольку класс `Vector` реализует интерфейс `List`, вектор можно применять точно так же, как экземпляр `ArrayList`. Вдобавок им можно манипулировать, используя его унаследованные методы. Скажем, после создания экземпляра `Vector` к нему можно добавить элемент, вызвав метод `addElement()`. Для получения элемента в определенной позиции понадобится вызвать `elementAt()`. Чтобы получить первый элемент вектора, нужно вызвать метод `firstElement()`. Чтобы получить последний элемент, необходимо вызвать метод `lastElement()`. С применением методов `indexOf()` и `lastIndexOf()` можно получить индекс элемента. Для удаления элемента потребуется вызвать метод `removeElement()` или `removeElementAt()`.

В следующей программе вектор используется для хранения различных типов числовых объектов. В ней демонстрируется несколько унаследованных методов, определенных в классе `Vector`, а также интерфейс `Enumeration`.

```
// Демонстрация разнообразных операций класса Vector.
import java.util.*;

class VectorDemo {
  public static void main(String[] args) {

    // Начальный размер равен 3, приращение равно 2.
    Vector<Integer> v = new Vector<Integer>(3, 2);

    System.out.println("Начальный размер: " + v.size());
    System.out.println("Начальная емкость: " + v.capacity());

    v.addElement(1);
    v.addElement(2);
    v.addElement(3);
    v.addElement(4);
```

```
    System.out.println("Емкость после четырех добавлений: " +
                       v.capacity());

    v.addElement(5);
    System.out.println("Текущая емкость: " +
                       v.capacity());

    v.addElement(6);
    v.addElement(7);

    System.out.println("Текущая емкость: " +
                       v.capacity());

    v.addElement(9);
    v.addElement(10);

    System.out.println("Текущая емкость: " +
                       v.capacity());

    v.addElement(11);
    v.addElement(12);

    System.out.println("Первый элемент: " + v.firstElement());
    System.out.println("Последний элемент: " + v.lastElement());

    if(v.contains(3))
      System.out.println("Вектор содержит элемент 3.");

    // Выполнить перечисление элементов в векторе.
    Enumeration<Integer> vEnum = v.elements();

    System.out.println("\nЭлементы в векторе:");
    while(vEnum.hasMoreElements())
      System.out.print(vEnum.nextElement() + " ");
    System.out.println();
  }
}
```

Ниже показан вывод программы:

```
Начальный размер: 0
Начальная емкость: 3
Емкость после четырех добавлений: 5
Текущая емкость: 5
Текущая емкость: 7
Текущая емкость: 9
Первый элемент: 1
Последний элемент: 12
Вектор содержит элемент 3.

Элементы в векторе:
1 2 3 4 5 6 7 9 10 11 12
```

Вместо того чтобы полагаться на перечисление для прохода по объектам (как делалось в предыдущей программе), можно воспользоваться итератором. Например, в программу можно подставить следующий код, основанный на итераторе:

```
// Использовать итератор для отображения содержимого.
Iterator<Integer> vItr = v.iterator();

System.out.println("\nЭлементы в векторе:");
```

```
while(vItr.hasNext())
  System.out.print(vItr.next() + " ");
System.out.println();
```

Для прохода по вектору можно также применять цикл for в стиле "for-each":

```
// Использовать расширенный цикл for для отображения содержимого.
System.out.println("\nЭлементы в векторе:");
for(int i : v)
  System.out.print(i + " ");
System.out.println();
```

Поскольку использовать интерфейс Enumeration в новом коде не рекомендуется, для перечисления содержимого вектора обычно будет применяться итератор или цикл for в стиле "for-each". Разумеется, в унаследованном коде используется Enumeration. К счастью, перечисления и итераторы работают практически одинаково.

Класс Stack

Класс Stack является подклассом Vector и реализует стандартный стек, функционирующий по принципу "последним пришел — первым обслужен". В Stack определен только стандартный конструктор, который создает пустой стек. В версии JDK 5 класс Stack был модифицирован с учетом обобщений и объявлен, как показано ниже:

```
class Stack<E>
```

В E указывается тип элементов, сохраняемых в стеке.

Класс Stack включает в себя все методы, определенные в Vector, и добавляет несколько собственных, которые кратко описаны в табл. 20.21.

Таблица 20.21. Методы, определенные в классе Stack

Метод	Описание
boolean empty()	Возвращает true, если стек пуст, или false, если он содержит элементы
E peek()	Возвращает элемент из верхушки стека, но не удаляет его
E pop()	Возвращает элемент из верхушки стека, удаляя его в процессе
E push(E element)	Помещает в стек элемент, указанный в element, и возвращает этот элемент
int search(Object element)	Ищет в стеке элемент, указанный в element. Если элемент найден, тогда возвращается его смещение от верхушки стека. В противном случае возвращается -1

Чтобы поместить объект на верхушку стека, необходимо вызвать метод `push()`. Чтобы удалить и возвратить верхний элемент, нужно вызвать метод `pop()`. Метод `peek()` можно применять для возвращения верхнего объекта без его удаления. В случае вызова `pop()` или `peek()`, когда вызывающий стек пуст, генерируется исключение `EmptyStackException`. Метод `empty()` возвращает `true`, если в стеке ничего нет. Метод `search()` определяет, существует ли объект в стеке, и возвращает количество вызовов метода `pop()`, требуемых для его перемещения на верхушку стека. В следующем примере создается стек, куда помещается несколько объектов `Integer` и затем извлекаются оттуда:

```
// Демонстрация использования класса Stack.
import java.util.*;

class StackDemo {
  static void showpush(Stack<Integer> st, int a) {
    st.push(a);
    System.out.println("push(" + a + ")");
    System.out.println("стек: " + st);
  }
  static void showpop(Stack<Integer> st) {
    System.out.print("pop -> ");
    Integer a = st.pop();
    System.out.println(a);
    System.out.println("стек: " + st);
  }
  public static void main(String[] args) {
    Stack<Integer> st = new Stack<Integer>();
    System.out.println("стек: " + st);
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
    showpop(st);
    showpop(st);
    showpop(st);
    try {
      showpop(st);
    } catch (EmptyStackException e) {
      System.out.println("стек пуст");
    }
  }
}
```

Ниже приведен вывод, генерируемый программой; обратите внимание на использование обработчика исключений `EmptyStackException` для изящной обработки ситуации с пустым стеком:

```
стек: []
push(42)
стек: [42]
push(66)
стек: [42, 66]
push(99)
стек: [42, 66, 99]
pop -> 99
```

```
стек: [42, 66]
pop -> 66
стек: [42]
pop -> 42
стек: []
pop -> стек пуст
```

Еще один момент: хотя класс Stack не объявлен нерекомендуемым, более удачным вариантом будет класс ArrayDeque.

Класс Dictionary

Dictionary — это абстрактный класс, представляющий хранилище для ключей и значений, который работает аналогично классу Map. При наличии ключа и значения значение можно хранить в объекте Dictionary. После сохранения значение можно получать с применением его ключа. Таким образом, подобно карте словарь можно рассматривать как список пар "ключ-значение". Несмотря на то что класс Dictionary в настоящее время не объявлен нерекомендуемым, он считается устаревшим, т.к. полностью замещен классом Map. Тем не менее, Dictionary все еще используется и потому здесь обсуждается. В версии JDK 5 класс Dictionary стал обобщенным. Вот его объявление:

```
class Dictionary<K, V>
```

В K указывается тип ключей, а в V — тип значений. Абстрактные методы, определенные в классе Dictionary, перечислены в табл. 20.22.

Таблица 20.22. Абстрактные методы, определенные в классе Dictionary

Метод	Описание
`Enumeration<V> elements()`	Возвращает перечисление значений, содержащихся в словаре
`V get (Object key)`	Возвращает объект, который содержит значение, ассоциированное с ключом key. Если ключ отсутствует в словаре, тогда возвращается null
`boolean isEmpty()`	Возвращает true, если словарь пуст, или false, если словарь содержит хотя бы один ключ
`Enumeration<K> keys()`	Возвращает перечисление ключей, содержащихся в словаре
`V put(K key, V value)`	Вставляет в словарь ключ, указанный в key, и его значение, заданное в value. Возвращает null, если ключ отсутствует в словаре, или предыдущее значение, ассоциированное с ключом, если ключ имеется в словаре
`V remove (Object key)`	Удаляет ключ, указанный в key, и его значение. Возвращает значение, ассоциированное с ключом. Если ключ отсутствует в словаре, тогда возвращается null
`int size()`	Возвращает количество элементов в словаре

Для добавления ключа и значения применяйте метод put(). Для получения значения заданного ключа используйте метод get(). Возвратить ключи и значения в виде перечисления можно с помощью методов keys() и elements(). Метод size() возвращает количество пар "ключ-значение", хранящихся в словаре, а метод isEmpty() возвращает true, если словарь пуст. Удалить пару "ключ-значение" позволяет метод remove().

Помните! Класс Dictionary объявлен устаревшим. Чтобы получить функциональность хранилища ключей и значений, потребуется реализовать интерфейс Map.

Класс Hashtable

Класс Hashtable был частью первоначального пакета java.util и представлял собой конкретную реализацию абстрактного класса Dictionary. Однако с появлением коллекций класс Hashtable был переработан с целью реализации интерфейса Map, что позволило интегрировать его в инфраструктуру Collections Framework. Он похож на HashMap, но является синхронизированным.

Как и HashMap, класс Hashtable хранит пары "ключ-значение" в хеш-таблице. Тем не менее, ни ключи, ни значения не могут быть null. В случае применения Hashtable должен указываться объект, служащий ключом, и значение, которое необходимо связать с этим ключом. Затем ключ хешируется, и полученный хеш-код используется в качестве индекса, по которому значение сохраняется в хеш-таблице.

В версии JDK 5 класс Hashtable был сделан обобщенным и получил следующее объявление:

```
class Hashtable<K, V>
```

В K указывается тип ключей, а в V — тип значений.

Хеш-таблица способна хранить только те ключи, для которых переопределены методы hashCode() и equals() из класса Object. Метод hashCode() должен вычислить и возвратить хеш-код для объекта. Конечно же, метод equals() сравнивает два объекта. К счастью, многие встроенные классы Java уже реализуют метод hashCode(). Так, для ключа в классе Hashtable выбран объект String. В классе String реализован как метод hashCode(), так и метод equals(). Ниже перечислены конструкторы класса Hashtable:

```
Hashtable()
Hashtable(int size)
Hashtable(int size, float fillRatio)
Hashtable(Map<? extends K, ? extends V> m)
```

Первая форма — стандартный конструктор. Вторая форма создает хеш-таблицу, начальный размер которой указывается в size. (Размер по умолчанию равен 11 элементов.) Третья форма создает хеш-таблицу с начальным размером, заданным в size, и коэффициентом заполнения, указанным в fillRatio. Коэффициент заполнения (он же коэффициент загрузки) должен находиться в диапазоне от 0.0 до 1.0 и определяет, насколько полной может

быть хеш-таблица, прежде чем ее размер будет увеличен. В частности, когда количество элементов превышает емкость хеш-таблицы, умноженную на ее коэффициент заполнения, хеш-таблица расширяется. Если коэффициент заполнения не указан, тогда применяется 0.75. Наконец, четвертая форма создает хеш-таблицу, которая инициализируется элементами карты m. По умолчанию используется коэффициент загрузки 0.75.

Помимо методов, определенных в интерфейсе Map, который класс Hashtable теперь реализует, в Hashtable имеются устаревшие методы, кратко описанные в табл. 20.23. Некоторые методы генерируют исключение NullPointerException при попытке применения ключа или значения null.

Таблица 20.23. Унаследованные методы, определенные в классе Hashtable

Метод	Описание
`void clear()`	Возвращает в исходное состояние и опустошает хеш-таблицу
`Object clone()`	Возвращает дубликат вызывающего объекта
`boolean contains (Object value)`	Возвращает true, если внутри хеш-таблицы существует значение, равное value, или false, если такое значение не найдено
`boolean containsKey (Object key)`	Возвращает true, если внутри хеш-таблицы существует ключ, равный key, или false, если такой ключ не найден
`boolean containsValue (Object value)`	Возвращает true, если внутри хеш-таблицы существует значение, равное value, или false, если такое значение не найдено
`Enumeration<V> elements()`	Возвращает перечисление значений, содержащихся в хеш-таблице
`V get(Object key)`	Возвращает объект, который содержит значение, ассоциированное с ключом key. Если ключ отсутствует в хеш-таблице, тогда возвращается null
`boolean isEmpty()`	Возвращает true, если хеш-таблица пуста, или false, если хеш-таблица содержит хотя бы один ключ
`Enumeration<K> keys()`	Возвращает перечисление ключей, содержащихся в хеш-таблице
`V put(K key, V value)`	Вставляет в хеш-таблицу ключ, указанный в key, и его значение, заданное в value. Возвращает null, если ключ отсутствует в хеш-таблице, или предыдущее значение, ассоциированное с ключом, если ключ имеется в хеш-таблице

Окончание табл. 20.23

Метод	Описание
`void rehash()`	Увеличивает размер хеш-таблицы и заново хеширует все ключи
`V remove (Object key)`	Удаляет ключ, указанный в key, и его значение. Возвращает значение, ассоциированное с ключом. Если ключ отсутствует в хеш-таблице, тогда возвращается null
`int size()`	Возвращает количество элементов в хеш-таблице
`String toString()`	Возвращает строковый эквивалент хеш-таблицы

В следующем примере показанная ранее программа, работающая с банковскими счетами, переделана так, чтобы для хранения имен клиентов банка и текущих балансов их счетов использовалась хеш-таблица:

```
// Демонстрация использования Hashtable.
import java.util.*;

class HTDemo {
  public static void main(String[] args) {
    Hashtable<String, Double> balance =
      new Hashtable<String, Double>();

    Enumeration<String> names;
    String str;
    double bal;

    balance.put("John Doe", 3434.34);
    balance.put("Tom Smith", 123.22);
    balance.put("Jane Baker", 1378.00);
    balance.put("Tod Hall", 99.22);
    balance.put("Ralph Smith", -19.08);

    // Отобразить балансы всех счетов из хеш-таблицы.
    names = balance.keys();
    while(names.hasMoreElements()) {
      str = names.nextElement();
      System.out.println(str + ": " +
                         balance.get(str));
    }

    System.out.println();

    // Пополнить счет клиента John Doe на 1000.
    bal = balance.get("John Doe");
    balance.put("John Doe", bal+1000);
    System.out.println("Новый баланс клиента John Doe: " +
                       balance.get("John Doe"));
  }
}
```

Вот вывод программы:

```
Todd Hall: 99.22
Ralph Smith: -19.08
John Doe: 3434.34
Jane Baker: 1378.0
Tom Smith: 123.22

Новый баланс клиента John Doe: 4434.34
```

Следует отметить один важный момент: как и классы карт, `Hashtable` не поддерживает итераторы напрямую, поэтому в предыдущей программе для отображения содержимого `balance` применяется перечисление. Однако можно получить представления хеш-таблицы в виде наборов и затем использовать итераторы, для чего вызвать один из методов представлений в виде коллекции из класса `Map` вроде `entrySet()` или `keySet()`. Например, можно получить представление в виде набора для ключей и проходить по нему с помощью итератора или расширенного цикла `for`. Вот переработанная версия программы, демонстрирующая такой прием:

```
// Использование итераторов с Hashtable.
import java.util.*;

class HTDemo2 {
  public static void main(String[] args) {
    Hashtable<String, Double> balance =
      new Hashtable<String, Double>();

    String str;
    double bal;

    balance.put("John Doe", 3434.34);
    balance.put("Tom Smith", 123.22);
    balance.put("Jane Baker", 1378.00);
    balance.put("Tod Hall", 99.22);
    balance.put("Ralph Smith", -19.08);

    // Отобразить балансы всех счетов из хеш-таблицы.
    // Для начала получить представление в виде набора для ключей.
    Set<String> set = balance.keySet();

    // Получить итератор.
    Iterator<String> itr = set.iterator();
    while(itr.hasNext()) {
      str = itr.next();
      System.out.println(str + ": " +
                         balance.get(str));
    }

    System.out.println();

    // Пополнить счет клиента John Doe на 1000.
    bal = balance.get("John Doe");
    balance.put("John Doe", bal+1000);
    System.out.println("Новый баланс клиента John Doe: " +
                       balance.get("John Doe"));
  }
}
```

Класс Properties

Класс Properties является подклассом Hashtable и применяется для поддержки списков значений, в которых ключ и значение относятся к типу String. Он используется рядом других классов Java. Например, типом объекта, возвращаемого методом System.getProperties() при получении значений переменных среды, будет как раз класс Properties. Сам по себе класс Properties не объявлен обобщенным, но некоторые из его методов определены как обобщенные. В классе Properties определена следующая изменчивая защищенная переменная экземпляра:

```
Properties defaults;
```

Переменная defaults содержит список стандартных свойств, ассоциированный с объектом Properties. В классе Properties определены перечисленные ниже конструкторы:

```
Properties()
Properties(Properties propDefault)
Properties(int capacity)
```

Первая форма конструктора создает объект Properties без стандартных значений. Вторая форма создает объект со стандартными значениями propDefault. В обоих случаях список свойств пуст. Третья форма позволяет указать начальную емкость для списка свойств. Во всех случаях список будет увеличиваться по мере необходимости.

В дополнение к методам, которые класс Properties наследует от Hashtable, в нем определены методы, кратко описанные в табл. 20.24. Кроме того, класс Properties содержит один нерекомендуемый метод: save(). Он был заменен методом store(), т.к. метод save() некорректно обрабатывал ошибки.

Таблица 20.24. Методы, определенные в классе Properties

Метод	Описание
String getProperty (String key)	Возвращает значение, ассоциированное с ключом key. Если ключ отсутствует в вызывающем списке и в стандартном списке свойств, тогда возвращается null
String getProperty (String key, String defaultProperty)	Возвращает значение, ассоциированное с ключом key. Если ключ отсутствует в вызывающем списке и в стандартном списке свойств, тогда возвращается defaultProperty
void list(PrintStream streamOut)	Направляет список свойств в поток вывода, связанный с тем, что указан в streamOut
void list(PrintWriter streamOut)	Направляет список свойств в поток вывода, связанный с тем, что указан в streamOut

Окончание табл. 20.24

Метод	Описание
`void load(InputStream streamIn) throws IOException`	Принимает список свойств из потока ввода, связанного с тем, что указан в `streamIn`
`void load(Reader streamIn) throws IOException`	Принимает список свойств из потока ввода, связанного с тем, что указан в `streamIn`
`void loadFromXML (InputStream streamIn) throws IOException, InvalidProperties-FormatException`	Принимает список свойств из XML-документа, связанного с тем, что указан в `streamIn`
`Enumeration<?> propertyNames()`	Возвращает перечисление ключей, куда также входят ключи, найденные в стандартном списке свойств
`Object setProperty (String key, String value)`	Ассоциирует значение `value` с ключом `key`. Возвращает предыдущее значение, ассоциированное с ключом, или `null`, если такая ассоциация не существует
`void store (OutputStream streamOut, String description) throws IOException`	Записывает в поток вывода `streamOut` строку, указанную в `description`, и затем список свойств
`void store (Writer streamOut, String description) throws IOException`	Записывает в поток вывода `streamOut` строку, указанную в `description`, и затем список свойств
`void storeToXML (OutputStream streamOut, String description) throws IOException`	Записывает в поток вывода `streamOut` список свойств и строку, указанную в `description`, в виде XML-документа
`void storeToXML (OutputStream streamOut, String description, String enc)`	Записывает в поток вывода `streamOut` список свойств и строку, указанную в `description`, в виде XML-документа с использованием заданной кодировки символов
`void storeToXML (OutputStream streamOut, String description, Charset cs)`	Записывает в поток вывода `streamOut` список свойств и строку, указанную в `description`, в виде XML-документа с использованием заданной кодировки символов
`Set<String> stringPropertyNames()`	Возвращает набор ключей

Класс Properties обладает удобной возможностью указания стандартного свойства, которое будет возвращено, если с определенным ключом не ассоциировано какое-либо значение. Например, стандартное значение можно указывать вместе с ключом в вызове метода getProperty(), например, getProperty("имя", "стандартное значение"). Если значение "имя" не найдено, тогда возвращается "стандартное значение". При создании объекта Properties можно передать еще один экземпляр Properties, который будет использоваться в качестве стандартных свойств для нового экземпляра. Тогда в случае вызова getProperty("foo") для заданного объекта Properties, если значение "foo" не существует, то оно ищется в стандартном объекте Properties. Допускается произвольное вложение уровней стандартных свойств.

В следующем примере демонстрируется применение класса Properties. В нем создается список свойств, в котором ключами являются названия штатов, а значениями — названия их столиц. Обратите внимание, что в попытку найти столицу Флориды включается стандартное значение.

```
// Демонстрация работы со списком свойств.
import java.util.*;

class PropDemo {
  public static void main(String[] args) {
    Properties capitals = new Properties();
    capitals.setProperty("Иллинойс", "Спрингфилд");
    capitals.setProperty("Миссури", "Джефферсон-Сити");
    capitals.setProperty("Вашингтон", "Олимпия");
    capitals.setProperty("Калифорния", "Сакраменто");
    capitals.setProperty("Индиана", "Индианаполис");

    // Получить представление в виде набора для ключей.
    Set<?> states = capitals.keySet();

    // Отобразить все штаты и их столицы.
    for(Object name : states)
      System.out.println("Столица штата " +
                         name + " - " +
                         capitals.getProperty((String)name)
                         + ".");
    System.out.println();

    // Найти штат, отсутствующий в списке - указать стандартное значение.
    String str = capitals.getProperty("Флорида", "не найдена");
    System.out.println("Столица штата Флорида - " + str + ".");
  }
}
```

Ниже показан вывод, генерируемый программой:

```
Столица штата Миссури - Джефферсон-Сити.
Столица штата Иллинойс - Спрингфилд.
Столица штата Индиана - Индианаполис.
Столица штата Калифорния - Сакраменто.
Столица штата Вашингтон - Олимпия.

Столица штата Флорида - не найдена.
```

Поскольку штат Флорида в списке отсутствует, используется стандартное значение.

Хотя, как было показано в предыдущем примере, применять стандартное значение при вызове `getProperty()` вполне допустимо, для большинства приложений со списками свойств существует более эффективный способ поддержки стандартных значений. Ради обеспечения большей гибкости при конструировании объекта `Properties` необходимо указать стандартный список свойств, в котором будет производиться поиск, когда нужный ключ не найден в основном списке. Например, ниже представлена слегка переработанная версия предыдущей программы с заданным стандартным списком штатов. Теперь при поиске штата Флорида он будет найден в стандартном списке:

```
// Использование стандартного списка свойств.
import java.util.*;

class PropDemoDef {
  public static void main(String[] args) {
    Properties defList = new Properties();
    defList.setProperty("Флорида", "Таллахасси");
    defList.setProperty("Висконсин", "Мадисон");

    Properties capitals = new Properties(defList);

    capitals.setProperty("Иллинойс", "Спрингфилд");
    capitals.setProperty("Миссури", "Джефферсон-Сити");
    capitals.setProperty("Вашингтон", "Олимпия");
    capitals.setProperty("Калифорния", "Сакраменто");
    capitals.setProperty("Индиана", "Индианаполис");

    // Получить представление в виде набора для ключей.
    Set<?> states = capitals.keySet();

    // Отобразить все штаты и их столицы.
    for(Object name : states)
      System.out.println("Столица штата " +
                         name + " - " +
                         capitals.getProperty((String)name)
                         + ".");

    System.out.println();

    // Штат Флорида теперь будет найден в стандартном списке.
    String str = capitals.getProperty("Флорида");
    System.out.println("Столица штата Флорида - " + str + ".");
  }
}
```

Использование методов `store()` и `load()`

Один из наиболее полезных аспектов класса `Properties` связан с тем, что информацию, содержащуюся в объекте `Properties`, можно легко сохранять и загружать с диска посредством методов `store()` и `load()`. Объект `Properties` можно в любой момент записать в поток и прочитать его обратно. В итоге списки свойств особенно удобны для реализации простых баз данных. Например, в приведенной далее программе список свойств применяется

для создания простого телефонного справочника, в котором хранятся имена и номера телефонов абонентов. Чтобы найти номер абонента, необходимо ввести его имя. Методы `store()` и `load()` используются для сохранения и извлечения списка. После запуска программа сначала пробует загрузить список из файла `phonebook.dat`. Если этот файл существует, тогда список загружается. Затем в список можно добавлять элементы и новый список будет сохранен при завершении работы программы. Обратите внимание, насколько мало кода требуется для реализации небольшого, но функционального телефонного справочника.

```
/*Простая база данных телефонных номеров, которая использует список свойств.*/
import java.io.*;
import java.util.*;

class Phonebook {
  public static void main(String[] args) throws IOException
  {
    Properties ht = new Properties();
    BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in, System.console().charset()));
    String name, number;
    FileInputStream fin = null;
    boolean changed = false;

    // Попробовать открыть файл phonebook.dat.
    try {
      fin = new FileInputStream("phonebook.dat");
    } catch(FileNotFoundException e) {
      // Игнорировать отсутствие файла.
    }

    /* Если файл phonebook.dat существует, тогда загрузить из него номера
телефонов. */
    try {
      if(fin != null) {
        ht.load(fin);
        fin.close();
      }
    } catch(IOException e) {
      System.out.println("Ошибка при чтении файла.");
    }

    // Предоставить пользователю возможность вводить новые имена и номера.
    do {
      System.out.println("Введите новое имя" +
                         " (quit для завершения): ");
      name = br.readLine();
      if(name.equals("quit")) continue;

      System.out.println("Введите номер: ");
      number = br.readLine();

      ht.setProperty(name, number);
      changed = true;
    } while(!name.equals("quit"));
```

```
    // Если данные телефонного справочника изменились, тогда сохранить его.
    if(changed) {
      FileOutputStream fout = new FileOutputStream("phonebook.dat");
      ht.store(fout, "Телефонный справочник");
      fout.close();
    }

    // Искать номера по заданному имени.
    do {
      System.out.println("Введите интересующее имя" +
                         " (quit для завершения): ");
      name = br.readLine();
      if(name.equals("quit")) continue;

      number = (String) ht.get(name);
      System.out.println(number);
    } while(!name.equals("quit"));
  }
}
```

Заключительные соображения по поводу коллекций

Инфраструктура Collections Framework предоставляет мощный набор продуманных решений для ряда наиболее распространенных задач программирования. Обязательно рассмотрите возможность использования коллекций, когда у вас возникнет потребность в хранении и извлечении информации. Не забывайте, что коллекции предназначены не только для решения "крупных задач", таких как корпоративные базы данных, списки рассылки или системы инвентаризации. Они также эффективны при применении к задачам меньшего размера. Скажем, класс `TreeMap` может послужить великолепной коллекцией для хранения структуры каталогов в рамках множества файлов. Класс `TreeSet` может оказаться весьма полезным для хранения информации по управлению проектом. Откровенно говоря, виды задач, которые выигрывают от решения на основе коллекций, ограничены лишь вашим воображением. И последнее замечание: в главе 30 обсуждается потоковый API. Поскольку потоки интегрированы с коллекциями, при работе с коллекцией обдумайте возможность использования потока.

ГЛАВА

21 Пакет java.util, часть 2: дополнительные служебные классы

Обсуждение пакета java.util в этой главе продолжается исследованием классов и интерфейсов, не входящих в состав инфраструктуры Collections Framework. К ним относятся классы, которые поддерживают таймеры, работают с датами, вычисляют случайные числа и упаковывают ресурсы. Кроме того, рассматриваются классы Formatter и Scanner, облегчающие запись и чтение форматированных данных, и класс Optional, упрощающий обработку ситуаций, когда значение может отсутствовать. В конце главы подведены итоги касательно подпакетов java.util. Особый интерес представляет подпакет java.util.function, в котором определено несколько стандартных функциональных интерфейсов. И последнее, что следует отметить: интерфейс Observer и класс Observable из java.util в JDK 9 были объявлены нерекомендуемыми и потому здесь они не обсуждаются.

Класс StringTokenizer

Обработка текста часто предусматривает синтаксический разбор форматированной входной строки. *Синтаксический разбор* — это разделение текста на набор отдельных частей, или *лексем*, которые в определенной последовательности могут передавать семантический смысл. Класс StringTokenizer обеспечивает первый шаг в процессе синтаксического разбора и его часто называют *лексическим анализатором* или *сканером*. Класс StringTokenizer реализует интерфейс Enumeration. Таким образом, с помощью StringTokenizer можно организовать перечисление содержащихся во входной строке лексем. Прежде чем начать, важно отметить, что класс StringTokenizer описан здесь в первую очередь для тех программистов, которые работают с унаследованным кодом. Более современную альтернативу для нового кода предлагают регулярные выражения, обсуждаемые в главе 31.

Для использования класса StringTokenizer необходимо указать входную строку и строку, содержащую разделители. Разделители — это символы, отделяющие лексемы друг от друга. Каждый символ в строке разделителей считается допустимым разделителем, например, ",;:" устанавливает в качестве разделителей запятую, точку с запятой и двоеточие. Стандартный набор раз-

делителей состоит из пробельных символов: пробела, табуляции, перевода страницы, новой строки и возврата каретки.

Ниже перечислены конструкторы класса StringTokenizer:

```
StringTokenizer(String str)
StringTokenizer(String str, String delimiters)
StringTokenizer(String str, String delimiters, boolean delimAsToken)
```

Во всех конструкторах в str передается строка, подлежащая лексическому анализу. В первой форме применяются стандартные разделители. Во второй и третьей формах конструктора в delimiters указывается строка с разделителями. Если delimAsToken в третьей форме конструктора имеет значение true, тогда при разборе строки разделители тоже возвращаются в виде лексем. В противном случае разделители не возвращаются. Первые две формы не возвращают разделители как лексемы.

После создания объекта StringTokenizer метод nextToken() используется для извлечения последовательных лексем. Метод hasMoreTokens() возвращает true, если существуют лексемы для извлечения.

Поскольку StringTokenizer реализует интерфейс Enumeration, методы hasMoreElements() и nextElement() тоже реализованы и действуют так же, как соответственно hasMoreTokens() и nextToken(). Методы класса StringTokenizer кратко описаны в табл. 21.1.

Таблица 21.1. Методы, объявленные в классе StringTokenizer

Метод	Описание
`int countTokens()`	С использованием текущего набора разделителей метод определяет количество лексем, оставшихся для разбора, и возвращает результат
`boolean hasMoreElements()`	Возвращает true, если в строке осталась одна или большее количество лексем, или false, если лексем больше нет
`boolean hasMoreTokens()`	Возвращает true, если в строке осталась одна или большее количество лексем, или false, если лексем больше нет
`Object nextElement()`	Возвращает следующую лексему в виде экземпляра Object
`String nextToken()`	Возвращает следующую лексему в виде экземпляра String
`String nextToken(String delimiters)`	Возвращает следующую лексему в виде экземпляра String и устанавливает строку с разделителями в delimiters

Ниже приведен пример создания экземпляра StringTokenizer для разбора пар "ключ=значение". Последовательно идущие наборы пар "ключ=значение" отделяются друг от друга точкой с запятой.

```
// Демонстрация работы StringTokenizer.
import java.util.StringTokenizer;

class STDemo {
  static String in = "title=Java: The Complete Reference;" +
                     "author=Schildt;" +
                     "publisher=McGraw Hill;" +
                     "copyright=2022";

public static void main(String[] args) {
   StringTokenizer st = new StringTokenizer(in, "=;");

   while(st.hasMoreTokens()) {
     String key = st.nextToken();
     String val = st.nextToken();
     System.out.println(key + "\t" + val);
   }
  }
}
```

Вот вывод, генерируемый программой:

```
title  Java: The Complete Reference
author  Schildt
publisher  McGraw Hill
copyright  2022
```

BitSet

Класс BitSet поддерживает создание особого типа массива, который содержит битовые значения в виде булевских величин. По мере необходимости массив может увеличиваться в размерах, что делает его похожим на вектор битов. Ниже перечислены конструкторы класса BitSet:

```
BitSet()
BitSet(int size)
```

Первая форма конструктора создает стандартный объект. Вторая форма позволяет указать его начальный размер (т.е. количество битов, которое он может вместить). Все биты инициализируются значением false.

В классе BitSet определены методы, кратко описанные в табл. 21.2.

Таблица 21.2. Методы, объявленные в классе BitSet

Метод	Описание
void and(BitSet bitSet)	Выполняет операцию логического И над содержимым вызывающего объекта BitSet и объекта, указанного в bitSet. Результат помещается в вызывающий объект

Продолжение табл. 21.2

Метод	Описание
`void andNot(BitSet bitSet)`	Для каждого установленного бита в `bitSet` очищается соответствующий бит в вызывающем объекте `BitSet`
`int cardinality()`	Возвращает количество установленных битов в вызывающем объекте
`void clear()`	Обнуляет все биты
`void clear(int index)`	Обнуляет бит, указанный с помощью `index`
`void clear(int startIndex, int endIndex)`	Обнуляет биты, начиная с позиции `startIndex` и заканчивая позицией `endIndex-1`
`Object clone()`	Дублирует вызывающий объект `BitSet`
`boolean equals(Object bitSet)`	Возвращает `true`, если вызывающий объект `BitSet` эквивалентен объекту, переданному в `bitSet`, или `false` в противном случае
`void flip(int index)`	Переключает бит, указанный в `index`
`void flip(int startIndex, int endIndex)`	Переключает биты, начиная с позиции `startIndex` и заканчивая позицией `endIndex-1`
`boolean get(int index)`	Возвращает текущее состояние бита, указанного в `index`
`BitSet get(int startIndex, int endIndex)`	Возвращает объект `BitSet`, который содержит биты, начиная с позиции `startIndex` и заканчивая позицией `endIndex-1`. Вызывающий объект не изменяется
`int hashCode()`	Возвращает хеш-код для вызывающего объекта
`boolean intersects(BitSet bitSet)`	Возвращает `true`, если установлена, по меньшей мере, одна пара соответствующих битов внутри вызывающего объекта и объекта `bitSet`
`boolean isEmpty()`	Возвращает `true`, если все биты вызывающего объекта очищены

Продолжение табл. 21.2

Метод	Описание
`int length()`	Возвращает количество битов, необходимое для хранения содержимого вызывающего объекта `BitSet`. Это значение определяется положением последнего установленного бита
`int nextClearBit(int startIndex)`	Возвращает индекс следующего очищенного бита (т.е. следующего бита, равного `false`), начиная с индекса `startIndex`
`int nextSetBit(int startIndex)`	Возвращает индекс следующего установленного бита (т.е. следующего бита, равного `true`), начиная с индекса `startIndex`. Если установленные биты отсутствуют, тогда возвращается `-1`
`void or(BitSet bitSet)`	Выполняет операцию логического ИЛИ над содержимым вызывающего объекта `BitSet` и объекта, указанного в `bitSet`. Результат помещается в вызывающий объект
`int previousClearBit(int startIndex)`	Возвращает индекс следующего очищенного бита (т.е. следующего бита, равного `false`) по индексу `startIndex` или перед ним. Если очищенные биты отсутствуют, тогда возвращается `-1`
`int previousSetBit(int startIndex)`	Возвращает индекс следующего установленного бита (т.е. следующего бита, равного `true`) по индексу `startIndex` или перед ним. Если установленные биты отсутствуют, тогда возвращается `-1`
`void set(int index)`	Устанавливает бит, указанный в `index`
`void set(int index, boolean v)`	Устанавливает бит, указанный в `index`, согласно значению, переданному в `v`. Значение `true` приводит к установке бита, а `false` — к его очистке
`void set(int startIndex, int endIndex)`	Устанавливает биты, начиная с позиции `startIndex` и заканчивая позицией `endIndex-1`

Окончание табл. 21.2

Метод	Описание
`void set(int startIndex, int endIndex, boolean v)`	Устанавливает биты, начиная с позиции `startIndex` и заканчивая позицией `endIndex-1`, согласно значению, переданному в `v`. Значение `true` приводит к установке битов, а `false` — к их очистке
`int size()`	Возвращает количество битов в вызывающем объекте `BitSet`
`IntStream stream()`	Возвращает поток данных, который содержит позиции установленных битов, от младшей до старшей
`byte[] toByteArray()`	Возвращает массив типа `byte`, который содержит вызывающий объект `BitSet`
`long[] toLongArray()`	Возвращает массив типа `long`, который содержит вызывающий объект `BitSet`
`String toString()`	Возвращает строковый эквивалент вызывающего объекта `BitSet`
`static BitSet valueOf(byte[] v)`	Возвращает объект `BitSet`, содержащий биты в `v`
`static BitSet valueOf(ByteBuffer v)`	Возвращает объект `BitSet`, содержащий биты в `v`
`static BitSet valueOf(long[] v)`	Возвращает объект `BitSet`, содержащий биты в `v`
`static BitSet valueOf(LongBuffer v)`	Возвращает объект `BitSet`, содержащий биты в `v`
`void xor(BitSet bitSet)`	Выполняет операцию логического исключающего ИЛИ над содержимым вызывающего объекта `BitSet` и объекта, указанного в `bitSet`. Результат помещается в вызывающий объект

Далее представлен пример применения `BitSet`:

```
// Демонстрация работы BitSet.
import java.util.BitSet;

class BitSetDemo {
  public static void main(String[] args) {
    BitSet bits1 = new BitSet(16);
    BitSet bits2 = new BitSet(16);
```

```
    // Установить ряд битов.
    for(int i=0; i<16; i++) {
      if((i%2) == 0) bits1.set(i);
      if((i%5) != 0) bits2.set(i);
    }

    System.out.println("Начальный набор битов в bits1: ");
    System.out.println(bits1);
    System.out.println("\nНачальный набор битов в bits2: ");
    System.out.println(bits2);

    // Выполнить операцию логического И с битами.
    bits2.and(bits1);
    System.out.println("\nbits2 AND bits1: ");
    System.out.println(bits2);

    // Выполнить операцию логического ИЛИ с битами.
    bits2.or(bits1);
    System.out.println("\nbits2 OR bits1: ");
    System.out.println(bits2);

    // Выполнить операцию логического исключающего ИЛИ с битами.
    bits2.xor(bits1);
    System.out.println("\nbits2 XOR bits1: ");
    System.out.println(bits2);
  }
}
```

Ниже показан вывод, генерируемый программой. Когда `toString()` преобразует объект `BitSet` в его строковый эквивалент, каждый установленный бит представляется своей битовой позицией. Очищенные биты не отображаются.

```
Начальный набор битов в bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Начальный набор битов в bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:
{}
```

Optional, OptionalDouble, OptionalInt и OptionalLong

Начиная с версии JDK 8, классы с именами `Optional`, `OptionalDouble`, `OptionalInt` и `OptionalLong` позволяют справиться с ситуациями, когда значение может присутствовать или отсутствовать. В прошлом для указания на то, что значение отсутствует, обычно использовалось значение `null`. Однако при попытке разыменования ссылки `null` может возникнуть исключение,

связанное с указателем на null. Во избежание генерации такого исключения требовались частые проверки на предмет null. Перечисленные классы обеспечивают лучший способ обработки таких случаев. И еще один момент: эти классы основаны на значениях. (Классы, основанные на значениях, были описаны в главе 13.)

Первым и наиболее универсальным из этих классов является Optional и потому он будет основным предметом обсуждения:

```
class Optional<T>
```

В T указывается тип хранимого значения. Важно понимать, что экземпляр Optional может либо содержать значение типа T, либо быть пустым. Другими словами, объект Optional не обязательно содержит значение. В классе Optional конструкторы не определены, но есть несколько методов, позволяющих работать с объектами Optional. Например, можно выяснить, присутствует ли значение, получить значение при его наличии, получить стандартное значение, когда значение отсутствует, и создать значение Optional. Методы класса Optional кратко описаны в табл. 21.3.

Таблица 21.3. Методы, объявленные в классе Optional

Метод	Описание
`static <T> Optional<T> empty()`	Возвращает объект, для которого метод `isPresent()` возвращает `false`
`boolean equals(Object optional)`	Возвращает `true`, если вызывающий объект равен `optional`, или `false` в противном случае
`Optional<T> filter(Predicate<? super T> condition)`	Возвращает экземпляр Optional, который содержит то же самое значение, что и вызывающий объект, если это значение удовлетворяет условию, указанному в condition. В противном случае возвращается пустой объект
`U Optional<U> flatMap (Function<? super T, Optional<U>> mapFunc)`	Применяет функцию сопоставления, указанную в mapFunc, к вызывающему объекту, если этот объект содержит значение, и возвращает результат. В противном случае возвращается пустой объект
`T get()`	Возвращает значение в вызывающем объекте. Если значение отсутствует, тогда генерируется исключение NoSuchElementException
`int hashCode()`	Возвращает хеш-код для значения в вызывающем объекте или 0, если значение отсутствует
`void ifPresent (Consumer<? super T> func)`	Вызывает func, если в вызывающем объекте присутствует значение, передавая объект в func. Если значение отсутствует, то никакие действия не предпринимаются

Продолжение табл. 21.3

Метод	Описание
`void ifPresentOrElse( Consumer<? super T> func, Runnable onEmpty)`	Вызывает func, если в вызывающем объекте присутствует значение, передавая объект в func. Если значение отсутствует, то выполняется onEmpty
`boolean isEmpty()`	Возвращает true, если вызывающий объект не содержит значение, или false, если значение присутствует
`boolean isPresent()`	Возвращает true, если вызывающий объект содержит значение, или false, если значение отсутствует
`U Optional<U> map (Function<? super T, ? extends U>> mapFunc)`	Применяет функцию сопоставления, указанную в mapFunc, к вызывающему объекту, если этот объект содержит значение, и возвращает результат. В противном случае возвращается пустой объект
`static <T> Optional<T> of(T val)`	Создает экземпляр Optional, содержащий val, и возвращает результат. Значением val не должно быть null
`static <T> Optional<T> ofNullable(T val)`	Создает экземпляр Optional, содержащий val, и возвращает результат. Если val равно null, тогда возвращается пустой экземпляр Optional
`Optional<T> or(Supplier<? extends Optional<? extends T>> func)`	Если в вызывающем объекте отсутствует какое-либо значение, тогда вызывает func для конструирования экземпляра Optional, который содержит значение. В противном случае возвращается экземпляр Optional, содержащий значение вызывающего объекта
`T orElse(T defVal)`	Если вызывающий объект содержит значение, то оно возвращается. В противном случае возвращается значение, указанное в defVal
`T orElseGet(Supplier<? extends T> getFunc)`	Если вызывающий объект содержит значение, то оно возвращается. В противном случае возвращается значение, полученное из getFunc
`T orElseThrow()`	Возвращает значение в вызывающем объекте. Если значение отсутствует, тогда генерируется исключение NoSuchElementException

Окончание табл. 21.3

Метод	Описание
`<X extends Throwable> T orElseThrow (Supplier<? extends X> excFunc) throws X extends Throwable`	Возвращает значение в вызывающем объекте. Если значение отсутствует, тогда инициируется исключение, сгенерированное excFunc
`Stream<T> stream()`	Возвращает поток данных, который содержит значение вызывающего объекта. Если значение отсутствует, тогда поток не будет содержать значений
`String toString()`	Возвращает строку, соответствующую вызывающему объекту

Лучший способ понять работу класса `Optional` — рассмотреть пример, в котором используются его ключевые методы. В основе `Optional` лежат `isPresent()` и `get()`. Выяснить, присутствует ли значение, можно с помощью метода `isPresent()`. Если значение доступно, тогда `isPresent()` возвратит `true`, а если нет, то `false`. Получить значение, существующее в экземпляре `Optional`, можно посредством вызова `get()`. Тем не менее, в случае вызова `get()` для объекта, который не содержит значение, генерируется исключение `NoSuchElementException`. По этой причине всегда необходимо сначала удостовериться в наличии значения, прежде чем вызывать `get()` на объекте `Optional`. Начиная с версии JDK 10, вместо `get()` можно использовать форму метода `orElseThrow()` без параметров, имя которого делает операцию более понятной. Однако в примерах в этой книге будет применяться `get()`, так что код будет компилироваться и у тех читателей, которые имеют дело с более ранними версиями Java.

Конечно, потребность в вызове двух методов для получения значения увеличивает накладные расходы при каждом доступе. К счастью, в классе `Optional` определены методы, сочетающие проверку и извлечение значения. Одним из таких методов является `orElse()`. Если объект, на котором он вызывается, содержит значение, то значение возвращается. В противном случае возвращается стандартное значение.

Конструкторы в классе `Optional` не определены. Взамен вы будете использовать один из его методов для создания экземпляра. Например, создать экземпляр `Optional` с указанным значением можно с применением метода `of()`, а экземпляр `Optional`, не содержащий значения — с помощью `empty()`. Упомянутые методы демонстрируются в следующей программе:

```
// Демонстрация работы нескольких методов Optional<T>.
import java.util.*;

class OptionalDemo {
```

```
  public static void main(String[] args) {
    Optional<String> noVal = Optional.empty();
    Optional<String> hasVal = Optional.of("ABCDEFG");

    if(noVal.isPresent()) System.out.println("Это не отобразится");
    else System.out.println("noVal не имеет значения");

    if(hasVal.isPresent()) System.out.println("Строка в hasVal: " +
                                              hasVal.get());
    String defStr = noVal.orElse("Стандартная строка");
    System.out.println(defStr);
  }
}
```

Вот вывод:

```
noVal не имеет значения
Строка в hasVal: ABCDEFG
Стандартная строка
```

В выводе видно, что значение можно получить из объекта `Optional`, только если оно присутствует. Такой базовый механизм позволяет классу `Optional` предотвращать возникновение исключений, связанных с указателем на `null`.

Классы `OptionalDouble`, `OptionalInt` и `OptionalLong` работают в основном подобно классу `Optional`, но разработаны специально для использования со значениями типа `double`, `int` и `long` соответственно. Таким образом, в них определены методы `getAsDouble()`, `getAsInt()` и `getAsLong()`, а не `get()`. Кроме того, они не поддерживают методы `filter()`, `ofNullable()`, `map()`, `flatMap()` и `or()`.

Date

Класс `Date` инкапсулирует текущую дату и время. Прежде чем приступить к его изучению, важно отметить, что он существенно изменился по сравнению с исходной версией, определенной в Java 1.0. Когда была выпущена версия Java 1.1, многие функции, выполняемые исходным классом `Date`, были перемещены в классы `Calendar` и `DateFormat`, и в результате многие методы `Date` исходной версии 1.0 были объявлены нерекомендуемыми. Поскольку устаревшие методы версии 1.0 не должны применяться в новом коде, здесь они не рассматриваются.

Класс `Date` поддерживает следующие конструкторы, не являющиеся нерекомендуемыми:

```
Date()
Date(long millisec)
```

Первая форма конструктора инициализирует объект текущей датой и временем. Вторая форма конструктора принимает аргумент с количеством милисекунд, прошедших с 1 января 1970 года. В табл. 21.4 кратко описаны методы класса `Date`, которые не объявлены нерекомендуемыми. Кроме того, класс `Date` реализует интерфейс `Comparable`.

Таблица 21.4. Методы класса Date, которые не объявлены нерекомендуемыми

Метод	Описание
`boolean after(Date date)`	Возвращает `true`, если вызывающий объект `Date` содержит более позднюю дату, чем дата, указанная в `date`, или `false` в противном случае
`boolean before(Date date)`	Возвращает `true`, если вызывающий объект `Date` содержит более раннюю дату, чем дата, указанная в `date`, или `false` в противном случае
`Object clone()`	Дублирует вызывающий объект `Date`
`int compareTo(Date date)`	Сравнивает значение вызывающего объекта со значением `date`. Возвращает `0`, если даты равны, отрицательное значение, если вызывающий объект содержит более раннюю дату, чем `date`, и положительное значение, если вызывающий объект содержит более позднюю дату, чем `date`
`boolean equals(Object date)`	Возвращает `true`, если вызывающий объект `Date` содержит то же самое время и дату, что и указанные в `date`, или `false` в противном случае
`static Date from(Instant t)`	Возвращает объект `Date`, который соответствует объекту `Instant`, переданному в `t`
`long getTime()`	Возвращает количество миллисекунд, прошедших с 1 января 1970 года
`int hashCode()`	Возвращает хеш-код для вызывающего объекта
`void setTime(long time)`	Устанавливает время и дату, как указано в `time`, что представляет количество миллисекунд, прошедших с 1 января 1970 года
`Instant toInstant()`	Возвращает объект `Instant`, соответствующий вызывающему объекту `Date`
`String toString()`	Преобразует вызывающий объект `Date` в строку и возвращает результат

В табл. 21.4 видно, что методы Date, которые не объявлены нерекомендуемыми, не позволяют получать индивидуальные компоненты даты или времени. Как демонстрируется в следующей программе, получить дату и время можно только в миллисекундах, в стандартном строковом представлении, возвращаемом toString(), или в виде объекта Instant. Для получения более подробной информации о дате и времени предназначен класс Calendar.

```
// Отображение даты и времени с использованием только методов класса Date.
import java.util.Date;

class DateDemo {
  public static void main(String[] args) {
    // Создать объект Date.
    Date date = new Date();
    // Отобразить время и дату с применением toString().
    System.out.println(date);
    // Отобразить количество миллисекунд, прошедших с 1 января 1970 года,
    // как среднее время по Гринвичу.
    long msec = date.getTime();
    System.out.println("Количество миллисекунд, прошедших с 1 января 1970 года,
GMT = " + msec);
  }
}
```

Ниже показан пример вывода:

```
Sat Jan 01 10:52:44 CST 2022
Количество миллисекунд, прошедших с 1 января 1970 года,
GMT = 1641056951341
```

Calendar

Абстрактный класс Calendar предлагает набор методов, позволяющих преобразовывать время в миллисекундах в несколько полезных компонентов, скажем, в год, месяц, день, часы, минуты и секунды. Предполагается, что подклассы Calendar будут предоставлять специальную функциональность для интерпретации информации о времени в соответствии с собственными правилами. Это один из аспектов библиотеки классов Java, который делает возможным написание программ, способных работать в международных средах. Примером такого подкласса является GregorianCalendar.

На заметку! Еще один API-интерфейс для работы с датой и временем находится в пакете java.time (см. главу 31).

Класс Calendar не предоставляет открытых конструкторов. В Calendar определен ряд защищенных переменных экземпляра. Давайте взглянем на них. Переменная areFieldsSet представляет собой булевское значение, которое указывает, установлены ли компоненты времени. Переменная fields — целочисленный массив, содержащий компоненты времени. Переменная isSet — булевский массив, который указывает, установлен ли конкретный компонент времени. Переменная time — значение типа long, соответствую-

щее текущему времени для данного объекта. Переменная isTimeSet — булевское значение, которое указывает, установлено ли текущее время.

В табл. 21.5 кратко описаны избранные методы класса Calendar.

Таблица 21.5. Избранные методы, определенные в классе Calendar

Метод	Описание
`abstract void add(int which, int val)`	Добавляет значение val к компоненту времени или даты, заданному в which. Для вычитания необходимо добавить отрицательное значение. В which должно быть указано одно из полей, определенных в Calendar, например, Calendar.HOUR
`boolean after(Object calendarObj)`	Возвращает true, если вызывающий объект Calendar содержит более позднюю дату, чем та, что указана в calendarObj, или false в противном случае
`boolean before(Object calendarObj)`	Возвращает true, если вызывающий объект Calendar содержит более раннюю дату, чем та, что указана в calendarObj, или false в противном случае
`final void clear()`	Обнуляет все компоненты времени в вызывающем объекте
`final void clear(int which)`	Обнуляет компонент времени, указанный с помощью which, в вызывающем объекте
`Object clone()`	Возвращает дубликат вызывающего объекта
`boolean equals(Object calendarObj)`	Возвращает true, если вызывающий объект Calendar содержит дату, равную указанной в calendarObj, или false в противном случае
`int get(int calendarField)`	Возвращает значение одного компонента вызывающего объекта. Компонент указывается в calendarField. Примерами запрашиваемых компонентов могут служить Calendar.YEAR, Calendar.MONTH, Calendar.MINUTE и т.д.
`static Locale[] getAvailableLocales()`	Возвращает массив объектов Locale, содержащих локали, для которых доступны календари
`static Calendar getInstance()`	Возвращает объект Calendar для стандартной локали и часового пояса

Окончание табл. 21.5

Метод	Описание
`static Calendar getInstance(TimeZone tz)`	Возвращает объект `Calendar` для часового пояса, указанного в `tz`. Используется стандартная локаль
`static Calendar getInstance (Locale locale)`	Возвращает объект `Calendar` для локали, указанной в `locale`. Используется стандартный часовой пояс
`static Calendar getInstance(TimeZone tz, Locale locale)`	Возвращает объект `Calendar` для часового пояса, указанного в `tz`, и локали, заданной в `locale`
`final Date getTime()`	Возвращает объект `Date`, эквивалентный времени в вызывающем объекте
`TimeZone getTimeZone()`	Возвращает часовой пояс для вызывающего объекта
`final boolean isSet(int which)`	Возвращает `true`, если указанный в `which` компонент времени установлен, или `false` в противном случае
`void set(int which, int val)`	Устанавливает в вызывающем объекте компонент даты или времени, указанный с помощью `which`, в значение, заданное посредством `val`. В `which` должно быть указано одно из полей, определенных в `Calendar`, такое как `Calendar.HOUR`
`final void set(int year, int month, int dayOfMonth)`	Устанавливает различные компоненты даты и времени в вызывающем объекте
`final void set(int year, int month, int dayOfMonth, int hours, int minutes)`	Устанавливает различные компоненты даты и времени в вызывающем объекте
`final void set(int year, int month, int dayOfMonth, int hours, int minutes, int seconds)`	Устанавливает различные компоненты даты и времени в вызывающем объекте
`final void setTime(Date d)`	Устанавливает различные компоненты даты и времени в вызывающем объекте. Информация получается из объекта `d` типа `Date`
`void setTimeZone(TimeZone tz)`	Устанавливает для вызывающего объекта часовой пояс, указанный в `tz`
`final Instant toInstant()`	Возвращает объект `Instant`, соответствующий вызывающему объекту `Calendar`

В классе Calendar определены перечисленные ниже константы типа int, которые используются при получении или установке компонентов календаря.

ALL_STYLES	HOUR_OF_DAY	PM
AM	JANUARY	SATURDAY
AM_PM	JULY	SECOND
APRIL	JUNE	SEPTEMBER
AUGUST	LONG	SHORT
DATE	LONG_FORMAT	SHORT_FORMAT
DAY_OF_MONTH	LONG_STANDALONE	SHORT_STANDALONE
DAY_OF_WEEK	MARCH	SUNDAY
DAY_OF_WEEK_IN_MONTH	MAY	THURSDAY
DAY_OF_YEAR	MILLISECOND	TUESDAY
DECEMBER	MINUTE	UNDECIMBER
DST_OFFSET	MONDAY	WEDNESDAY
ERA	MONTH	WEEK_OF_MONTH
FEBRUARY	NARROW_FORMAT	WEEK_OF_YEAR
FIELD_COUNT	NARROW_STANDALONE	YEAR
FRIDAY	NOVEMBER	ZONE_OFFSET
HOUR	OCTOBER	

В следующей программе демонстрируется работа нескольких методов класса Calendar:

```
// Демонстрация работы нескольких методов класса Calendar.
import java.util.Calendar;

class CalendarDemo {
  public static void main(String[] args) {
    String[] months = {
      "Jan", "Feb", "Mar", "Apr",
      "May", "Jun", "Jul", "Aug",
      "Sep", "Oct", "Nov", "Dec"};

    // Создать календарь, инициализированный текущей датой и временем
    // в стандартной локали и часовом поясе.
    Calendar calendar = Calendar.getInstance();

    // Отобразить текущую информацию о времени и дате.
    System.out.print("Дата: ");
    System.out.print(months[calendar.get(Calendar.MONTH)]);
    System.out.print(" " + calendar.get(Calendar.DATE) + " ");
    System.out.println(calendar.get(Calendar.YEAR));

    System.out.print("Время: ");
    System.out.print(calendar.get(Calendar.HOUR) + ":");
    System.out.print(calendar.get(Calendar.MINUTE) + ":");
    System.out.println(calendar.get(Calendar.SECOND));
```

```
    // Установить информацию о времени и дате и отобразить ее.
    calendar.set(Calendar.HOUR, 10);
    calendar.set(Calendar.MINUTE, 29);
    calendar.set(Calendar.SECOND, 22);
    System.out.print("Обновленное время: ");
    System.out.print(calendar.get(Calendar.HOUR) + ":");
    System.out.print(calendar.get(Calendar.MINUTE) + ":");
    System.out.println(calendar.get(Calendar.SECOND));
  }
}
```

Вот пример вывода:

```
Дата: Jan 1 2022
Время: 11:29:39
Обновленное время: 10:29:22
```

GregorianCalendar

Класс GregorianCalendar является конкретной реализацией традиционного григорианского календаря с помощью Calendar. Метод getInstance() класса Calendar обычно возвращает GregorianCalendar, инициализированный текущей датой и временем в стандартной локали и часовом поясе.

В классе GregorianCalendar определены два поля, AD и BC, которые представляют две эпохи, определенные григорианским календарем.

Существует также несколько конструкторов для объектов GregorianCalendar. Конструктор GregorianCalendar() инициализирует объект текущей датой и временем в стандартной локали и часовом поясе. Остальные три конструктора обладают повышенным уровнем специализации:

```
GregorianCalendar(int year, int month, int dayOfMonth)
GregorianCalendar(int year, int month, int dayOfMonth,
                  int hours, int minutes)
GregorianCalendar(int year, int month, int dayOfMonth,
                  int hours, int minutes, int seconds)
```

Все три формы устанавливают день, месяц и год. Год указывается в year. Месяц задается в month, причем 0 соответствует январю. День месяца определяется в dayOfMonth. Первая форма конструктора устанавливает время в полночь. Вторая форма также устанавливает часы и минуты. В третьей форме добавлены секунды.

Кроме того, создать объект GregorianCalendar можно с указанием локали и/или часового пояса. Следующие конструкторы создают объекты, инициализированные текущей датой и временем в заданном часовом поясе и/или локали:

```
GregorianCalendar(Locale locale)
GregorianCalendar(TimeZone timeZone)
GregorianCalendar(TimeZone timeZone, Locale locale)
```

Класс GregorianCalendar обеспечивает реализацию всех абстрактных методов класса Calendar, а также предоставляет ряд дополнительных методов. Вероятно, наиболее интересным из них является isLeapYear(), который проверяет, является ли год високосным:

```
boolean isLeapYear(int year)
```

Метод isLeapYear() возвращает true, если год является високосным, или false в противном случае. Есть еще два интересных метода, from() и toZonedDateTime(), которые поддерживают API-интерфейс даты и времени, добавленный JDK 8 и упакованный в java.time.

Работа класса GregorianCalendar демонстрируется в следующей программе:

```
// Демонстрация использования класса GregorianCalendar.
import java.util.*;

class GregorianCalendarDemo {
  public static void main(String[] args) {
    String[] months = {
      "Jan", "Feb", "Mar", "Apr",
      "May", "Jun", "Jul", "Aug",
      "Sep", "Oct", "Nov", "Dec"};
    int year;

    // Создать григорианский календарь, инициализированный текущей датой
    // и временем в стандартной локали и часовом поясе.
    GregorianCalendar gcalendar = new GregorianCalendar();

    // Отобразить текущую информацию о времени и дате.
    System.out.print("Дата: ");
    System.out.print(months[gcalendar.get(Calendar.MONTH)]);
    System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
    System.out.println(year = gcalendar.get(Calendar.YEAR));

    System.out.print("Время: ");
    System.out.print(gcalendar.get(Calendar.HOUR) + ":");
    System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
    System.out.println(gcalendar.get(Calendar.SECOND));

    // Проверить, является ли текущий год високосным.
    if(gcalendar.isLeapYear(year)) {
      System.out.println("Текущий год является високосным.");
    }
    else {
      System.out.println("Текущий год не является високосным.");
    }
  }
}
```

Ниже приведен пример вывода:

```
Дата: Jan 1 2022
Время: 1:45:5
Текущий год не является високосным.
```

TimeZone

С показаниями времени связан еще один класс — TimeZone. Абстрактный класс TimeZone позволяет работать со смещением часовых поясов относительно среднего времени по Гринвичу (Greenwich mean time — GMT), которое также называется универсальным скоординированным временем (Coordinated Universal Time — UTC). Вдобавок он вычисляет летнее время. Класс TimeZone предоставляет только стандартный конструктор.

В табл. 21.6 кратко описаны избранные методы класса TimeZone.

Таблица 21.6. Избранные методы, определенные в классе TimeZone

Метод	Описание
`Object clone()`	Возвращает версию `clone()`, специфичную для TimeZone
`static String[] getAvailableIDs()`	Возвращает массив объектов String, представляющий имена всех часовых поясов
`static String[] getAvailableIDs (int timeDelta)`	Возвращает массив объектов String, представляющий имена всех часовых поясов, которые имеют смещение timeDelta относительно GMT
`static TimeZone getDefault()`	Возвращает объект TimeZone, который представляет стандартный часовой пояс, используемый на хост-компьютере
`String getID()`	Возвращает имя вызывающего объекта TimeZone
`abstract int getOffset (int era, int year, int month, int dayOfMonth, int dayOfWeek, int millisec)`	Возвращает смещение, которое должно быть добавлено к GMT для вычисления локального времени. Это значение корректируется с учетом перехода на летнее время. Параметры метода представляют компоненты даты и времени
`abstract int getRawOffset()`	Возвращает низкоуровневое смещение (в миллисекундах), которое должно быть добавлено к GMT для вычисления локального времени. Это значение не корректируется с учетом перехода на летнее время
`static TimeZone getTimeZone(String tzName)`	Возвращает объект TimeZone для часового пояса по имени tzName
`abstract boolean inDaylightTime(Date d)`	Возвращает true, если дата, представленная с помощью d, находится в летнем времени в вызывающем объекте, или false в противном случае

Окончание табл. 21.6

Метод	Описание
`static void setDefault(TimeZone tz)`	Устанавливает стандартный часовой пояс, который будет использоваться на этом хост-компьютере. В `tz` передается ссылка на применяемый объект `TimeZone`
`void setID(String tzName)`	Устанавливает имя часового пояса (т.е. его идентификатор) в то, что указано в `tzName`
`abstract void setRawOffset(int millis)`	Устанавливает смещение в миллисекундах относительно GMT
`ZoneId toZoneId()`	Преобразует вызывающий объект в `ZoneId` и возвращает результат. Класс `ZoneId` находится в пакете `java.time`
`abstract boolean useDaylightTime()`	Возвращает `true`, если вызывающий объект использует переход на летнее время, или `false` в противном случае

SimpleTimeZone

Класс `SimpleTimeZone` является удобным подклассом `TimeZone`. Он реализует абстрактные методы `TimeZone` и позволяет работать с часовыми поясами для григорианского календаря, а также вычисляет летнее время.

В `SimpleTimeZone` определены четыре конструктора. Вот первый из них:

```
SimpleTimeZone(int timeDelta, String tzName)
```

Этот конструктор создает объект `SimpleTimeZone`. Смещение относительно GMT указывается в `timeDelta`; имя часового пояса задается в `tzName`.

Ниже показан второй конструктор `SimpleTimeZone`:

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
               int dstDayInMonth0, int dstDay0, int time0,
               int dstMonth1, int dstDayInMonth1, int dstDay1,
               int time1)
```

Смещение относительно GMT указывается в `timeDelta`. Имя часового пояса передается в `tzId`. Начало перехода на летнее время задается параметрами `dstMonth0`, `dstDayInMonth0`, `dstDay0` и `time0`. Конец летнего времени указывается в параметрах `dstMonth1`, `dstDayInMonth1`, `dstDay1` и `time1`.

А вот третий конструктор `SimpleTimeZone`:

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
               int dstDayInMonth0, int dstDay0, int time0,
               int dstMonth1, int dstDayInMonth1,
               int dstDay1, int time1, int dstDelta)
```

В dstDelta задается количество миллисекунд, сберегаемых в течение летнего времени.

Наконец, четвертый конструктор SimpleTimeZone выглядит так:

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
               int dstDayInMonth0, int dstDay0, int time0,
               int time0mode, int dstMonth1, int dstDayInMonth1,
               int dstDay1, int time1, int time1mode, int dstDelta)
```

В time0mode указывается режим начального времени, а в time1mode — режим конечного времени. Допустимые значения режима включают:

```
STANDARD_TIME          WALL_TIME          UTC_TIME
```

Режим времени устанавливает, каким образом интерпретируются значения времени. Остальные конструкторы используют стандартный режим WALL_TIME.

Locale

Экземпляр класса Locale создается для получения объектов, описывающих географический или культурный регион. Он относится к тем нескольким классам, которые дают возможность писать программы, способные выполняться в различных международных средах. Например, в разных регионах применяются отличающиеся форматы для отображения даты, времени и чисел.

Интернационализация — обширная тема, выходящая за рамки настоящей книги. Тем не менее, во многих программах придется иметь дело только с основами интернационализации, что предусматривает настройку текущей локали.

В классе Locale определены перечисленные далее константы, соответствующие распространенным локалям:

CANADA	GERMAN	KOREAN
CANADA_FRENCH	GERMANY	PRC
CHINA	ITALIAN	SIMPLIFIED_CHINESE
CHINESE	ITALY	TAIWAN
ENGLISH	JAPAN	TRADITIONAL_CHINESE
FRANCE	JAPANESE	UK
FRENCH	KOREA	US

Например, выражение Locale.CANADA представляет объект Locale для Канады.

Вот конструкторы класса Locale:

```
Locale(String language)
Locale(String language, String country)
Locale(String language, String country, String variant)
```

Показанные конструкторы создают объект Locale для представления конкретного языка, указанного в language, и в случае двух последних конструкторов — страны, заданной в country. Значения language и country должны содержать стандартные коды языка и страны.

Параметр variant позволяет предоставлять вспомогательную информацию. В классе Locale определено несколько методов. Одним из самых важных является setDefault():

```
static void setDefault(Locale localeObj)
```

Метод setDefault() устанавливает стандартную локаль, используемую машиной JVM, в ту, что указана в localeObj.

Ниже перечислены другие интересные методы класса Locale:

```
final String getDisplayCountry()
final String getDisplayLanguage()
final String getDisplayName()
```

Они возвращают удобочитаемые строки, которые можно применять для отображения названия страны, имени языка и полного описания локали.

Получить стандартную локаль можно с использованием метода getDefault():

```
static Locale getDefault()
```

В версии JDK 7 класс Locale был значительно обновлен и стал поддерживать спецификацию BCP 47 инженерной группы по развитию Интернета (Internet Engineering Task Force — IETF), которая определяет дескрипторы для идентификации языков, и технический стандарт Unicode (Unicode Technical Standard) под названием UTS 35, определяющий язык разметки данных локалей (Locale Data Markup Language — LDML). Поддержка BCP 47 и UTS 35 привела к добавлению в Locale ряда функциональных средств, включая несколько новых методов и класс Locale.Builder. В число новых методов входит getScript(), получающий сценарий локали, и toLanguageTag(), который получает строку, содержащую дескриптор языка локали.

Класс Locale.Builder позволяет создавать экземпляры Locale, гарантируя тем самым, что спецификация локали правильно сформирована, как определено в BCP 47. (Конструкторы Locale не обеспечивают такой проверки.) Кроме того, в версии JDK 8 в класс Locale были добавлены новые методы, среди которых методы, поддерживающие фильтрацию, расширения и поиск. В версии JDK 9 появился метод getISOCountries(), который возвращает коллекцию кодов стран для заданного значения перечисления Locale.IsoCountryCode.

Примерами классов, которые работают с учетом локали, могут служить Calendar и GregorianCalendar. Классы DateFormat и SimpleDateFormat тоже зависят от локали.

Random

Класс Random поддерживает генератор псевдослучайных чисел. Они называются *псевдослучайными*, поскольку представляют собой просто равномерно распределенные последовательности. Начиная с версии JDK 17, класс Random реализует новый интерфейс RandomGenerator, который является стандартизированным интерфейсом для генераторов случайных значений.

В Random определены следующие конструкторы:

```
Random()
Random(long seed)
```

Первая форма конструктора создает генератор чисел, который использует достаточно уникальное начальное число. Вторая форма позволяет указать начальное число вручную.

Инициализация объекта Random начальным числом определяет стартовую точку для случайной последовательности. Если применить то же самое начальное число для инициализации другого объекта Random, то будет получена такая же случайная последовательность. Чтобы сгенерировать разные последовательности, понадобится указывать разные начальные числа. Один из способов предусматривает использование для начального числа текущего времени. Такой подход снижает возможность получения повторяющихся последовательностей.

В табл. 21.7 кратко описаны основные открытые методы, предлагаемые классом Random. Они были доступны в Random на протяжении многих лет (некоторые с момента выхода Java 1.0) и применяются весьма широко.

Таблица 21.7. Основные открытые методы, определенные в классе Random

Метод	Описание
`boolean nextBoolean()`	Возвращает следующее случайное число типа `boolean`
`void nextBytes (byte[] vals)`	Заполняет `vals` случайно сгенерированными значениями
`double nextDouble()`	Возвращает следующее случайное число типа `double`
`float nextFloat()`	Возвращает следующее случайное число типа `float`
`double nextGaussian()`	Возвращает следующее случайное число из нормального распределения
`int nextInt()`	Возвращает следующее случайное число типа `int`
`int nextInt(int n)`	Возвращает следующее случайное число типа `int` из диапазона от 0 до `n`
`long nextLong()`	Возвращает следующее случайное число типа `long`
`void setSeed (long newSeed)`	Устанавливает начальное число (т.е. стартовую точку для генератора случайных чисел) в то, что указано с помощью `newSeed`

Как видите, существует семь типов случайных чисел, которые можно извлекать из объекта Random. Случайные булевские значения доступны через метод `nextBoolean()`. Случайные байты можно получить, вызвав метод `nextBytes()`. Случайные целые числа можно извлечь с помощью метода

`nextInt()`. Случайные длинные целые числа можно получить посредством `nextLong()`. Методы `nextFloat()` и `nextDouble()` возвращают значения типа `float` и `double` в диапазоне от `0.0` до `1.0`. Наконец, метод `nextGaussian()` возвращает значение `double` со стандартным отклонением `1.0` от центральной точки `0.0` (то, что называется *кривой нормального распределения*).

В приведенном далее примере формируется последовательность с помощью метода `nextGaussian()`, для чего получаются 100 случайных значений из нормального распределения, которые затем усредняются. Кроме того, подсчитывается количество значений, попадающих в пределы двух стандартных отклонений с шагом `0.5` в положительную или отрицательную сторону. Результат графически отображается на экране в повернутом виде.

```
// Демонстрация случайных значений из нормального распределения.
import java.util.Random;
class RandDemo {
  public static void main(String[] args) {
    Random r = new Random();
    double val;
    double sum = 0;
    int[] bell = new int[10];
    for(int i=0; i<100; i++) {
      val = r.nextGaussian();
      sum += val;
      double t = -2;
      for(int x=0; x<10; x++, t += 0.5)
        if(val < t) {
          bell[x]++; break;
        }
    }
    System.out.println("Среднее для значений: " + (sum/100));
    // Отобразить кривую нормального распределения в повернутом виде.
    for(int i=0; i<10; i++) {
      for(int x=bell[i]; x>0; x--)
        System.out.print("*");
      System.out.println();
    }
  }
}
```

Ниже показан результат одного запуска программы. Как видите, получается колоколоподобное распределение чисел.

```
Среднее для значений: 0.0702235271133344
**
*******
******
***************
******************
*****************
************
**********
********
***
```

Полезно отметить, что в версии JDK 8 в классе Random появились три новых метода, которые поддерживают потоковый API (см. главу 30): doubles(), ints() и longs(). Каждый из них возвращает ссылку на поток данных, содержащий последовательность псевдослучайных значений указанного типа. Для каждого метода предусмотрено несколько перегруженных версий. Вот их простейшие формы:

```
DoubleStream doubles()
IntStream ints()
LongStream longs()
```

Метод doubles() возвращает поток данных, содержащий псевдослучайные значения типа double в диапазоне от 0.0 до 1.0. Метод ints() возвращает поток данных с псевдослучайными значениями типа int. Метод longs() возвращает поток данных, который содержит псевдослучайные значения типа long. Для этих трех методов возвращаемый поток данных фактически бесконечен. С каждым методом связано несколько перегруженных версий, которые позволяют указывать размер потока, источник и верхнюю границу.

Timer и TimerTask

Пакет java.util предлагает одно интересное и полезное средство — возможность запланировать выполнение задачи на какое-то время в будущем. Такое средство поддерживается классами Timer и TimerTask, с помощью которых можно создать поток, работающий в фоновом режиме и ожидающий в течение определенного времени. Когда наступает надлежащий момент, выполняется задача, связанная с этим потоком. Доступные параметры позволяют планировать повторное выполнение задачи и ее запуск на определенную дату. Хотя с использованием класса Thread всегда можно было вручную создать задачу, которая будет выполняться в определенное время, классы Timer и TimerTask значительно упрощают такой процесс.

Классы Timer и TimerTask работают вместе. Timer — это класс, предназначенный для планирования выполнения задачи. Запланированная задача должна быть экземпляром TimerTask. Таким образом, чтобы запланировать задачу, сначала необходимо создать объект TimerTask и затем запланировать его выполнение с помощью экземпляра Timer.

Класс TimerTask реализует интерфейс Runnable; таким образом, его можно применять для создания потока выполнения. Ниже показан его конструктор:

```
protected TimerTask()
```

В классе TimerTask определены методы, кратко описанные в табл. 21.8. Обратите внимание, что метод run() является абстрактным, т.е. его необходимо переопределять. Метод run(), определенный в интерфейсе Runnable, содержит код, который будет выполняться. Таким образом, самый простой способ создать задачу таймера предусматривает расширение класса TimerTask и переопределение метода run().

Таблица 21.8. Методы, определенные в классе `TimerTask`

Метод	Описание
`boolean cancel()`	Прекращает работу задачи. Возвращает `true`, если выполнение задачи прервано, или `false` в противном случае
`abstract void run()`	Содержит код для задачи таймера
`long scheduledExecutionTime()`	Возвращает время, на которое было запланировано последнее выполнение задачи

После того, как задача создана, ее выполнение планируется с помощью объекта типа `Timer`. Вот конструкторы класса `Timer`:

```
Timer()
Timer(boolean DThread)
Timer(String tName)
Timer(String tName, boolean DThread)
```

Первая форма конструктора создает объект `Timer`, который работает как обычный поток. Вторая форма использует поток демона, если `DThread` имеет значение `true`. Поток демона будет выполняться только до тех пор, пока продолжает выполняться остальная часть программы. Третья и четвертая формы конструктора позволяют указывать имя потока `Timer`. Методы, определенные в классе `Timer`, кратко описаны в табл. 21.9.

Таблица 21.9. Методы, определенные в классе `Timer`

Метод	Описание
`void cancel()`	Прекращает работу потока таймера
`int purge()`	Удаляет отмененные задачи из очереди таймера
`void schedule (TimerTask TTask, long wait)`	Планирует выполнение задачи `TTask` по истечении промежутка времени в миллисекундах, переданного в параметре `wait`
`void schedule (TimerTask TTask, long wait, long repeat)`	Планирует выполнение задачи `TTask` по истечении промежутка времени в миллисекундах, переданного в параметре `wait`. Затем выполнение задачи повторяется с интервалом в миллисекундах, который указывается в параметре `repeat`
`void schedule (TimerTask TTask, Date targetTime)`	Планирует выполнение задачи `TTask` в момент времени, указанный в `targetTime`

Окончание табл. 21.9

Метод	Описание
`void schedule (TimerTask TTask, Date targetTime, long repeat)`	Планирует выполнение задачи `TTask` в момент времени, заданный в `targetTime`. Затем выполнение задачи повторяется с интервалом в миллисекундах, который указывается в параметре `repeat`
`void scheduleAtFixedRate (TimerTask TTask, long wait, long repeat)`	Планирует выполнение задачи `TTask` по истечении промежутка времени в миллисекундах, переданного в параметре `wait`. Затем выполнение задачи повторяется с интервалом в миллисекундах, который указывается в параметре `repeat`. Момент времени каждого повторения задается относительно первого запуска, а не предыдущего. Таким образом, общая частота запусков является фиксированной
`void scheduleAtFixedRate (TimerTask TTask, Date targetTime, long repeat)`	Планирует выполнение задачи `TTask` в момент времени, заданный в `targetTime`. Затем выполнение задачи повторяется с интервалом в миллисекундах, который указывается в параметре `repeat`. Момент времени каждого повторения задается относительно первого запуска, а не предыдущего. Таким образом, общая частота запусков является фиксированной

После создания объекта `Timer` выполнение задачи планируется с помощью вызова на объекте `Timer` метода `schedule()`. Как было показано в табл. 21.9, существует несколько форм `schedule()`, которые позволяют планировать задачи различными способами.

В случае создания задачи, не являющейся демоном, по завершении программы понадобится вызывать метод `cancel()`, чтобы завершить задачу, иначе программа может "зависнуть" на какое-то время.

Применение классов `Timer` и `TimerTask` демонстрируется в приведенной ниже программе, где определяется задача таймера, метод `run()` которой отображает сообщение `"Выполняется задача таймера."`. Задача запланирована на запуск каждые полсекунды после начальной задержки в одну секунду.

```
// Демонстрация использования классов Timer и TimerTask.
import java.util.*;
class MyTimerTask extends TimerTask {
  public void run() {
    System.out.println("Выполняется задача таймера.");
  }
}
```

```
class TTest {
  public static void main(String[] args) {
    MyTimerTask myTask = new MyTimerTask();
    Timer myTimer = new Timer();
    /* Установить начальную задержку в одну секунду
       и затем повторять каждые полсекунды.
    */
    myTimer.schedule(myTask, 1000, 500);
    try {
      Thread.sleep(5000);
    } catch (InterruptedException exc) {}
    myTimer.cancel();
  }
}
```

Currency

Класс Currency инкапсулирует информацию о денежной единице. Конструкторы в нем не определены, а методы кратко описаны в табл. 21.10.

Таблица 21.10. Методы, определенные в классе Currency

Метод	Описание
`static Set<Currency> getAvailableCurrencies()`	Возвращает набор поддерживаемых денежных единиц
`String getCurrencyCode()`	Возвращает код (определенный стандартом ISO 4217), который описывает денежную единицу вызывающего объекта
`int getDefaultFractionDigits()`	Возвращает количество цифр после десятичной точки, которое обычно используется в денежной единице вызывающего объекта. Например, в случае доллара принято указывать две цифры после десятичной точки
`String getDisplayName()`	Возвращает название денежной единицы вызывающего объекта для стандартной локали
`String getDisplayName(Locale loc)`	Возвращает название денежной единицы вызывающего объекта для локали, указанной в loc
`static Currency getInstance (Locale localeObj)`	Возвращает объект Currency для локали, указанной в localeObj
`static Currency getInstance (String code)`	Возвращает объект Currency, который ассоциирован с кодом денежной единицы, переданным в code

Окончание табл. 21.10

Метод	Описание
`int getNumericCode()`	Возвращает числовой код (определенный стандартом ISO 4217) для денежной единицы вызывающего объекта
`String getNumericCodeAsString()`	Возвращает строковую форму числового кода (определенного стандартом ISO 4217) для денежной единицы вызывающего объекта
`String getSymbol()`	Возвращает символ денежной единицы (вроде $) для вызывающего объекта
`String getSymbol (Locale localeObj)`	Возвращает символ денежной единицы (наподобие $) для локали, переданной в `localeObj`
`String toString()`	Возвращает код денежной единицы для вызывающего объекта

Использование класса `Currency` демонстрируется в следующей программе:

```
// Демонстрация использования класса Currency.
import java.util.*;

class CurDemo {
  public static void main(String[] args) {
    Currency c;

    c = Currency.getInstance(Locale.US);

    System.out.println("Символ: " + c.getSymbol());
    System.out.println("Стандартное количество цифр после десятичной
точки: " + c.getDefaultFractionDigits());
  }
}
```

Ниже показан вывод:

```
Символ: $
Стандартное количество цифр после десятичной точки: 2
```

Formatter

В основе поддержки Java создания форматированного вывода лежит класс `Formatter`. Он обеспечивает *преобразования форматов*, что позволяет отображать числа, строки, время и дату практически в любом удобном виде. Класс `Formatter` работает аналогично функции `printf()` в C/C++, поэтому если вы знакомы с языком C/C++, тогда научиться применять `Formatter` будет очень легко. Кроме того, дополнительно упрощается преобразование кода C/C++ в Java. Но даже если вы не знакомы с языком C или C++, то форматировать данные все равно довольно просто.

На заметку! Хотя класс `Formatter` в Java работает очень похоже на функцию `printf()` в C/C++, с ним связаны некоторые отличия и новые возможности. При наличии опыта работы с C/C++ рекомендуется внимательно прочитать дальнейший материал главы.

Конструкторы класса `Formatter`

Чтобы можно было использовать класс `Formatter` для форматирования вывода, необходимо создать объект `Formatter`. Класс `Formatter` работает путем преобразования двоичной формы данных, задействованных внутри программы, в форматированный текст. Он сохраняет форматированный текст в буфере, содержимое которого может быть в любой момент получено в программе. Можно позволить `Formatter` предоставить такой буфер автоматически или же указать буфер явно при создании объекта `Formatter`. Также можно заставить `Formatter` выводить свой буфер в файл.

В классе `Formatter` определено множество конструкторов, которые позволяют создавать объект `Formatter` разнообразными способами, например:

```
Formatter()
Formatter(Appendable buf)
Formatter(Appendable buf, Locale loc)
Formatter(String filename)
  throws FileNotFoundException
Formatter(String filename, String charset)
  throws FileNotFoundException, UnsupportedEncodingException
Formatter(File outF)
  throws FileNotFoundException
Formatter(OutputStream outStrm)
```

В параметре `buf` указывается буфер для форматированного вывода. Если `buf` имеет значение `null`, тогда `Formatter` автоматически выделяет память под `StringBuilder` для хранения форматированного вывода. В параметре `loc` указывается локаль. Если локаль не задана, тогда применяется стандартная локаль. В параметре `filename` передается имя файла, который будет получать форматированный вывод. В параметре `charset` указывается набор символов. Если набор символов не задан, тогда используется стандартный набор символов. В параметре `outF` передается ссылка на открытый файл, который будет получать выходные данные. В параметре `outStrm` указывается ссылка на выходной поток, который будет получать выходные данные. В случае работы с файлом вывод тоже будет записываться в файл.

Пожалуй, наиболее широко применяемым конструктором является первый, который не имеет параметров. Он автоматически использует стандартную локаль и выделяет память под `StringBuilder` для хранения форматированного вывода.

Методы класса `Formatter`

В классе `Formatter` определены методы, которые кратко описаны в табл. 21.11.

Таблица 21.11. Методы, определенные в классе `Formatter`

Метод	Описание
`void close()`	Закрывает вызывающий объект `Formatter`, что приводит к освобождению всех ресурсов, используемых объектом. После того, как объект `Formatter` закрыт, применять его повторно нельзя. Попытка использования закрытого объекта `Formatter` дает в результате исключение `FormatterClosedException`
`void flush()`	Очищает буфер для форматированного вывода, что приводит к записи выходных данных, находящихся в текущий момент внутри буфера, в целевой объект. Метод применяется главным образом к объекту `Formatter`, привязанному к файлу
`Formatter format (String fmtString, Object ... args)`	Форматирует аргументы, переданные в `args`, согласно спецификаторам формата, содержащимся в `fmtString`. Возвращает вызывающий объект
`Formatter format (Locale loc, String fmtString, Object ... args)`	Форматирует аргументы, переданные в `args`, согласно спецификаторам формата, содержащимся в `fmtString`. При форматировании используется локаль, укащанная в `loc`. Возвращает вызывающий объект
`IOException ioException()`	Если объект, являющийся целью вывода, сгенерировал исключение `IOException`, тогда возвращает его, или `null` в противном случае
`Locale locale()`	Возвращает локаль вызывающего объекта
`Appendable out()`	Возвращает ссылку на объект, являющийся целью вывода
`String toString()`	Возвращает объект типа `String`, содержащий форматированный вывод

Основы форматирования

После создания объект `Formatter` можно применять для создания форматированной строки, используя его метод `format()`. Ниже показана версия, которая будет применяться:

```
Formatter format(String fmtString, Object ... args)
```

Строка `fmtSring` состоит из элементов двух типов. Первый тип содержит символы, которые просто копируются в выходной буфер, а второй тип включает спецификаторы формата, определяющие способ отображения последующих аргументов.

В своей простейшей форме спецификатор формата начинается со знака процента, за которым следует *спецификатор преобразования* формата. Все спецификаторы преобразования формата состоят из одного символа. Например, спецификатором формата для данных с плавающей точкой является `%f`. Как правило, аргументов должно быть столько же, сколько спецификаторов формата, а спецификаторы формата и аргументы сопоставляются в порядке слева направо. Например, рассмотрим приведенный далее фрагмент кода:

```
Formatter fmt = new Formatter();
fmt.format("Форматировать %s легко: %d %f", "с помощью Java", 10, 98.6);
```

Фрагмент кода создает объект `Formatter`, который содержит следующую строку:

```
Форматировать с помощью Java легко: 10 98.600000
```

В этом примере спецификаторы формата `%s`, `%d` и `%f` заменяются аргументами, которые следуют за строкой формата. Таким образом, `%s` заменяется на `"с помощью Java"`, `%d` — на `10`, а `%f` — на `98.6`. Все остальные символы используются в том виде как есть. Несложно догадаться, что спецификатор формата `%s` указывает строку, а `%d` — целочисленное значение. Как упоминалось ранее, `%f` задает значение с плавающей точкой.

Метод `format()` воспринимает множество спецификаторов формата, которые показаны в табл. 21.12. Обратите внимание, что многие спецификаторы записываются как в верхнем, так и в нижнем регистре. В случае применения спецификатора в верхнем регистре буквы отображаются в верхнем регистре. В остальном спецификаторы в верхнем и нижнем регистре выполняют одно и то же преобразование. Важно понимать, что Java проверяет тип каждого спецификатора формата на соответствие его аргументу. Если тип аргумента не совпадает, тогда генерируется исключение `IllegalFormatException`.

Таблица 21.12. Спецификаторы формата

Спецификатор формата	Применяемое преобразование
`%a` `%A`	Шестнадцатеричное число с плавающей точкой
`%b` `%B`	Булевское значение
`%c` `%C`	Символ
`%d`	Десятичное целое число
`%h` `%H`	Хеш-код аргумента

Окончание табл. 21.12

Спецификатор формата	Применяемое преобразование
%e %E	Экспоненциальная форма записи
%f	Десятичное число с плавающей точкой
%g %G	Используется спецификатор %e или %f в зависимости от форматируемого значения и точности
%o	Восьмеричное целое число
%n	Вставка символа новой строки
%s %S	Строка
%t %T	Время и дата
%x %X	Шестнадцатеричное целое число
%%	Вставка символа %

После форматирования строку можно получить, вызвав метод `toString()`. Продолжим предыдущий пример, получив форматированную строку, содержащуюся в `fmt`:

```
String str = fmt.toString();
```

Конечно, если нужно просто отобразить форматированную строку, то нет причин сначала присваивать ее объекту `String`. Например, при передаче объекта `Formatter` в `println()` автоматически вызывается его метод `toString()`.

Вот короткая программа, в которой все фрагменты собраны вместе для демонстрации создания и отображения форматированной строки:

```
// Очень простой пример использования Formatter.
import java.util.*;

class FormatDemo {
  public static void main(String[] args) {
    Formatter fmt = new Formatter();

    fmt.format("Форматировать %s легко: %d %f", "с помощью Java", 10, 98.6);
    System.out.println(fmt);
    fmt.close();
  }
}
```

Еще один момент: получить ссылку на внутренний выходной буфер можно с помощью вызова метода `out()`, который возвращает ссылку на объект `Appendable`.

Теперь, когда вам известен общий механизм создания форматированной строки, в оставшемся материале этого раздела подробно обсуждается каждое преобразование. Кроме того, будут описаны различные параметры, такие как выравнивание, минимальная ширина поля и точность.

Форматирование строк и символов

Для форматирования индивидуального символа используйте спецификатор `%c`, который приведет к выводу соответствующего символьного аргумента в неизмененном виде. Для форматирования строки применяйте `%s`.

Форматирование чисел

Для форматирования целого числа в десятичной форме записи используйте `%d`. Для форматирования числа с плавающей точкой в десятичной форме записи применяйте `%f`. Для форматирования числа с плавающей точкой в экспоненциальной форме записи используйте `%e`. Числа, представленные в экспоненциальной форме записи, в общем случае выглядят так:

```
x.dddddde+/-yy
```

Спецификатор формата `%g` заставляет объект `Formatter` применять `%f` или `%e` в зависимости от форматируемого значения и точности, которая по умолчанию равна 6. В следующей программе демонстрируется действие спецификаторов формата `%f` и `%e`:

```
// Демонстрация использования спецификаторов формата %f и %e.
import java.util.*;

class FormatDemo2 {
  public static void main(String[] args) {
    Formatter fmt = new Formatter();

    for(double i=1.23; i < 1.0e+6; i *= 100) {
      fmt.format("%f %e ", i, i);
      System.out.println(fmt);
    }
    fmt.close();
  }
}
```

Вот вывод:

```
1.230000 1.230000e+00
1.230000 1.230000e+00 123.000000 1.230000e+02
1.230000 1.230000e+00 123.000000 1.230000e+02 12300.000000 1.230000e+04
```

Целые числа можно отображать в восьмеричной или шестнадцатеричной форме записи, используя соответственно `%o` и `%x`. Например, показанный ниже фрагмент:

```
fmt.format("Шестнадцатеричное число: %x, восьмеричное число: %o", 196, 196);
```

производит такой вывод:

```
Шестнадцатеричное число: c4, восьмеричное число: 304
```

Значения с плавающей точкой можно отображать в шестнадцатеричной форме записи с применением `%a`. Формат, создаваемый `%a`, на первый взгляд кажется немного странным. Это связано с тем, что в его представлении используется форма, аналогичная экспоненциальной записи, которая состоит из шестнадцатеричной мантиссы и десятичной степени 2. Вот общий формат:

```
0x1.sigpexp
```

Здесь в `sig` содержится дробная часть мантиссы, `exp` — показатель степени, а `p` обозначает начало показателя степени. Скажем, следующий вызов:

```
fmt.format("%a", 512.0);
```

дает представленный далее вывод:

```
0x1.0p9
```

Форматирование времени и даты

Одним из наиболее мощных спецификаторов преобразования является `%t`, который позволяет форматировать информацию о времени и дате. Спецификатор `%t` работает немного иначе, чем другие, потому что он требует использования суффикса для описания части и точного формата желаемого времени или даты. Суффиксы кратко описаны в табл. 21.13. Например, для отображения минут применяется `%tM`, где `M` задает минуты в двухсимвольном поле. Аргумент, соответствующий спецификатору `%t`, должен иметь тип `Calendar`, `Date`, `Long`, `long` или `TemporalAccessor`.

Таблица 21.13. Суффиксы формата времени и даты

Суффикс	Чем заменяется
a	Сокращенное название дня недели
A	Полное название дня недели
b	Сокращенное название месяца
B	Полное название месяца
c	Стандартная строка с датой и временем в формате "день месяц дата чч::мм:сс часовой-пояс год"
C	Первые две цифры года
d	День месяца в виде десятичного числа (01–31)
D	месяц/день/год
e	День месяца в виде десятичного числа (1–31)
F	год-месяц-день
h	Сокращенное название месяца
H	Часы (00–23)
I	Часы (01–12)

Окончание табл. 21.13

Суффикс	Что заменяется
j	День года в виде десятичного числа (001–366)
k	Часы (0–23)
l	Часы (1–12)
L	Миллисекунды (000–999)
m	Месяц в виде десятичного числа (01–13)
M	Минуты в виде десятичного числа (00–59)
N	Наносекунды (000000000–999999999)
p	Эквивалент AM или PM для локали в нижнем регистре
Q	Количество миллисекунд, прошедших с 1 января 1970 года
r	чч:мм:сс (12-часовой формат)
R	чч:мм (24-часовой формат)
S	Секунды (00–60)
s	Количество секунд, прошедших с 1 января 1970 года UTC
T	чч:мм:сс (24-часовой формат)
y	Год в десятичном виде без столетия (00–99)
Y	Год в десятичном виде, включая столетие (0001–9999)
z	Смещение относительно UTC
Z	Имя часового пояса

Ниже приведена программа, в которой демонстрируется использование нескольких форматов времени и даты:

```
// Форматирование времени и даты.
import java.util.*;

class TimeDateFormat {
  public static void main(String[] args) {
    Formatter fmt = new Formatter();
    Calendar cal = Calendar.getInstance();

    // Отобразить время в стандартном 12-часовом формате.
    fmt.format("%tr", cal);
    System.out.println(fmt);
    fmt.close();

    // Отобразить полную информацию о дате и времени.
    fmt = new Formatter();
    fmt.format("%tc", cal);
    System.out.println(fmt);
    fmt.close();
```

```
    // Отобразить только часы и минуты.
    fmt = new Formatter();
    fmt.format("%tl:%tM", cal, cal);
    System.out.println(fmt);
    fmt.close();

    // Отобразить месяц по названию и номеру.
    fmt = new Formatter();
    fmt.format("%tB %tb %tm", cal, cal, cal);
    System.out.println(fmt);
    fmt.close();
  }
}
```

Так выглядит пример вывода:

```
03:15:34 PM
Sat Jan 01 15:15:34 CST 2022
3:15
January Jan 01
```

Спецификаторы %n и %%

Спецификаторы формата %n и %% отличаются от других тем, что они не сопоставляются с аргументом, а представляют собой просто управляющие последовательности, которые вставляют символ в выходную последовательность. Спецификатор %n вставляет символ новой строки. Спецификатор %% вставляет знак процента. Ни один из этих символов нельзя вводить напрямую в строке формата. Разумеется, для встраивания символа новой строки можно также применять стандартную управляющую последовательность \n.

Вот пример, в котором демонстрируется использование спецификаторов формата %n и %%:

```
// Демонстрация использования спецификаторов формата %n и %%.
import java.util.*;

class FormatDemo3 {
  public static void main(String[] args) {
    Formatter fmt = new Formatter();

    fmt.format("Копирование файла%nПередача завершена на %d%%", 88);
    System.out.println(fmt);
    fmt.close();
  }
}
```

Отображается следующий вывод:

```
Копирование файла
Передача завершена на 88%
```

Указание минимальной ширины поля

Целое число, помещенное между знаком % и кодом преобразования формата, действует как *спецификатор минимальной ширины поля*. В результате

вывод дополняется пробелами, обеспечивая достижение определенной минимальной ширины. Если строка или число длиннее минимальной ширины, то оно все равно будет выведено полностью. По умолчанию дополнение осуществляется пробелами. При желании дополнять нулями поместите 0 перед спецификатором ширины поля. Например, %05d дополнит число, состоящее менее чем из пяти цифр, нулями, так что его общая длина составит пять. Спецификатор ширины поля можно применять со всеми спецификаторами формата кроме %n.

В приведенной далее программе демонстрируется использование спецификатора минимальной ширины поля за счет его применения к преобразованию %f:

```
// Демонстрация использования спецификатора минимальной ширины поля.
import java.util.*;

class FormatDemo4 {
  public static void main(String[] args) {
    Formatter fmt = new Formatter();

    fmt.format("|%f|%n|%12f|%n|%012f|",
               10.12345, 10.12345, 10.12345);
    System.out.println(fmt);
    fmt.close();
  }
}
```

Программа генерирует такой вывод:

```
|10.123450|
|  10.123450|
|00010.123450|
```

В первой строке отображается число 10.12345 со стандартной шириной. Во второй строке то же самое значение отображается в 12-символьном поле. В третьей строке значение 10.12345 отображается в 12-символьном поле и дополняется начальными нулями.

Спецификатор минимальной ширины поля часто используется для создания таблиц, в которых колонки выравниваются одна под другой. Например, в следующей программе создается таблица квадратов и кубов для чисел от 1 до 10:

```
// Создание таблицы квадратов и кубов.
import java.util.*;

class FieldWidthDemo {
  public static void main(String[] args) {
    Formatter fmt;

    for(int i=1; i <= 10; i++) {
      fmt = new Formatter();
      fmt.format("%4d %4d %4d", i, i*i, i*i*i);
      System.out.println(fmt);
      fmt.close();
    }
  }
}
```

Ниже показан результирующий вывод:

```
 1       1       1
 2       4       8
 3       9      27
 4      16      64
 5      25     125
 6      36     216
 7      49     343
 8      64     512
 9      81     729
10     100    1000
```

Указание точности

Спецификатор точности может быть применен среди прочего к спецификаторам формата `%f`, `%e`, `%g` и `%s`. Он указывается после спецификатора минимальной ширины поля (при его наличии) и состоит из точки, за которой следует целое число. Его точный смысл зависит от типа данных, к которым он применяется. Когда спецификатор точности применяется к данным с плавающей точкой с использованием спецификаторов `%f` или `%e`, он определяет количество отображаемых десятичных разрядов. Например, `%10.4f` обеспечивает отображение числа шириной, по крайней мере, десять символов с четырьмя десятичными разрядами. В случае применения `%g` точность определяет количество значащих цифр. По умолчанию точность равна 6.

Применительно к строкам спецификатор точности задает максимальную ширину поля. Например, `%5.7s` позволяет отобразить строку длиной не менее пяти и не более семи символов. Если длина строки превышает максимальную ширину поля, то конечные символы будут усекаться.

Работа спецификатора точности иллюстрируется в следующей программе:

```
// Демонстрация использования спецификатора точности.
import java.util.*;
class PrecisionDemo {
  public static void main(String[] args) {
    Formatter fmt = new Formatter();
    // Форматировать с 4 десятичными разрядами.
    fmt.format("%.4f", 123.1234567);
    System.out.println(fmt);
    fmt.close();
    // Форматировать 2 десятичными разрядами в 16-символьном поле.
    fmt = new Formatter();
    fmt.format("%16.2e", 123.1234567);
    System.out.println(fmt);
    fmt.close();
    // Отобразить не более 15 символов в строке.
    fmt = new Formatter();
    fmt.format("%.15s", "Форматировать с помощью Java теперь легко.");
    System.out.println(fmt);
    fmt.close();
  }
}
```

Вот вывод:

```
123.1235
1.23e+02
Форматировать с
```

Использование флагов формата

Класс `Formatter` распознает набор флагов формата, которые позволяют управлять различными аспектами преобразования. Все флаги формата являются одиночными символами; каждый флаг формата следует за символом `%` в спецификации формата. Флаги формата перечислены в табл. 21.14.

Таблица 21.14. Флаги формата

Суффикс	Чем заменяется
`-`	Выравнивание по левому краю
#	Альтернативный формат преобразования
0	Выводимые данные дополняется нулями, а не пробелами
пробел	Положительные числа выводятся с предшествующим пробелом
+	Положительные числа выводятся с предшествующим знаком +
,	Числовые значения содержат разделители групп
(	Отрицательные числа заключаются в круглые скобки

Не все флаги применяются ко всем спецификаторам формата. Каждый флаг формата подробно рассматривается в одном из последующих разделов.

Выравнивание выводимых данных

По умолчанию все выводимые данные выравниваются по правому краю, т.е. если ширина поля больше длины выводимых данных, то они будут размещаться на правом краю поля. Поместив знак "минус" сразу после `%`, выводимые данные можно принудительно выровнять по левому краю. Например, `%-10.2f` обеспечивает выравнивание по левому краю числа с плавающей точкой с двумя десятичными разрядами в 10-символьном поле. Взгляните на следующую программу:

```
// Демонстрация выравнивания по левому краю.
import java.util.*;

class LeftJustify {
  public static void main(String[] args) {
    Formatter fmt = new Formatter();

    // По умолчанию выполняется выравнивание по правому краю.
    fmt.format("|%10.2f|", 123.123);
    System.out.println(fmt);
    fmt.close();
```

```
    // Теперь выровнять по левому краю.
    fmt = new Formatter();
    fmt.format("|%-10.2f|", 123.123);
    System.out.println(fmt);
    fmt.close();
  }
}
```

Ниже показан вывод:

```
|    123.12|
|123.12    |
```

Как видите, вторая строка выровнена по левому краю внутри 10-символьного поля.

Флаги пробела, +, 0 и (

Чтобы перед положительными числовыми значениями отображался знак +, необходимо добавить флаг +. Например, следующий оператор:

```
fmt.format("%+d", 100);
```

создает строку:

```
+100
```

При создании колонок чисел иногда полезно выводить пробел перед положительными значениями, чтобы положительные и отрицательные значения были выровненными. Для этого понадобится добавить флаг пробела. Ниже приведен пример:

```
// Демонстрация использования флага пробела.
import java.util.*;

class FormatDemo5 {
  public static void main(String[] args) {
    Formatter fmt = new Formatter();

    fmt.format("% d", -100);
    System.out.println(fmt);
    fmt.close();

    fmt = new Formatter();
    fmt.format("% d", 100);
    System.out.println(fmt);
    fmt.close();

    fmt = new Formatter();
    fmt.format("% d", -200);
    System.out.println(fmt);
    fmt.close();

    fmt = new Formatter();
    fmt.format("% d", 200);
    System.out.println(fmt);
    fmt.close();
  }
}
```

Вот вывод:

```
-100
 100
-200
 200
```

Обратите внимание, что положительные значения имеют начальный пробел, благодаря чему цифры в колонке выстраиваются правильно.

Для отображения выводимых отрицательных чисел в круглых скобках, а не с начальным знаком "минус", предназначен флаг (. Скажем, следующий оператор:

```
fmt.format("%(d", -100);
```

создает такую строку:

```
(100)
```

Флаг 0 приводит к тому, что выводимые данные дополняются нулями вместо пробелов.

Флаг запятой

При отображении крупных чисел часто полезно добавлять разделители групп, которыми в английском языке являются запятые. Например, значение 1234567 легче читать, если оно отформатировано как 1,234,567. Добавить разделители групп позволяет флаг запятой (,). Например, показанный далее оператор:

```
fmt.format("%,.2f", 4356783497.34);
```

создает следующую строку:

```
4,356,783,497.34
```

Флаг

Флаг # можно применять к спецификаторам %o, %x, %e и %f. Для %e и %f флаг # гарантирует наличие десятичной точки, даже если десятичных разрядов нет. Если перед спецификатором формата %x поместить #, тогда шестнадцатеричное число будет выводиться с префиксом 0x. Если перед спецификатором %o поместить #, то число будет выводиться с ведущим нулем.

Версии в верхнем регистре

Как упоминалось ранее, некоторые спецификаторы формата имеют версии в верхнем регистре, результатом действия которых будет использование преобразованием прописных букв там, где это необходимо. Версии спецификаторов в верхнем регистре кратко описаны в табл. 21.15.

Таблица 21.15. Версии спецификаторов в верхнем регистре

Спецификатор	Действие
%A	Приводит к тому, что шестнадцатеричные цифры a–f отображаются в верхнем регистре как A–F. Кроме того, префикс 0x отображается как 0X, а p — как P
%B	Приводит к отображению в верхнем регистре значений true и false
%E	Приводит к отображению в верхнем регистре символа e, обозначающего экспоненту
%G	Приводит к отображению в верхнем регистре символа e, обозначающего экспоненту
%H	Приводит к тому, что шестнадцатеричные цифры a–f отображаются в верхнем регистре как A–F
%S	Приводит к отображению в верхнем регистре соответствующей строки
%X	Приводит к тому, что шестнадцатеричные цифры a–f отображаются в верхнем регистре как A–F. Кроме того, необязательный префикс 0x отображается как 0X, если он присутствует

Например, следующий вызов:

```
fmt.format("%X", 250);
```

создает такую строку:

```
FA
```

А показанный ниже вызов:

```
fmt.format("%E", 123.1234);
```

создает строку вида:

```
1.231234E+02
```

Использование индекса аргумента

В состав класса `Formatter` входит очень полезное функциональное средство, позволяющее указывать аргумент, к которому применяется спецификатор формата. Обычно спецификаторы формата и аргументы сопоставляются по порядку слева направо, т.е. первый спецификатор формата соответствует первому аргументу, второй спецификатор — второму аргументу и т.д. Однако с использованием *индекса аргумента* можно явно управлять тем, какой аргумент соответствует определенному спецификатору формата.

Индекс аргумента указывается непосредственно после символа `%` в спецификаторе формата и имеет такую форму:

```
n$
```

где n — индекс желаемого аргумента, начинающийся с 1. Взгляните на следующий пример:

```
fmt.format("%3$d %1$d %2$d", 10, 20, 30);
```

Он создает строку вида:

```
30 10 20
```

В этом примере первый спецификатор формата сопоставляется с аргументом 30, второй — с аргументом 10, а третий — с аргументом 20. Таким образом, аргументы потребляются в порядке, отличающемся от строгого порядка слева направо.

Одно из преимуществ индексов аргументов заключается в том, что они позволяют повторно использовать аргумент без необходимости указывать его дважды. Например, рассмотрим показанный далее вызов:

```
fmt.format("%d в шестнадцатеричной форме записи выглядит как %1$x", 255);
```

Он создает следующую строку:

```
255 в шестнадцатеричной форме записи выглядит как ff
```

Как видите, аргумент 255 задействован в обоих спецификаторах формата.

Существует удобное сокращение под названием *относительный индекс*, которое позволяет повторно применять аргумент, соответствующий предыдущему спецификатору формата. Нужно просто указать < в качестве индекса аргумента. Скажем, приведенный ниже вызов `format()` дает те же результаты, что и в предыдущем примере:

```
fmt.format("%d в шестнадцатеричной форме записи выглядит как %<x", 255);
```

Относительные индексы особенно полезны при создании специальных форматов времени и даты. Рассмотрим следующий пример:

```
// Использование относительных индексов для упрощения
// создания специального формата времени и даты.
import java.util.*;

class FormatDemo6 {
  public static void main(String[] args) {
    Formatter fmt = new Formatter();
    Calendar cal = Calendar.getInstance();

    fmt.format("Сегодня день %te месяца %<tB, %<tY", cal);
    System.out.println(fmt);
    fmt.close();
  }
}
```

Вот вывод, полученный в результате одного из запусков:

```
Сегодня день 1 месяца January, 2022
```

Благодаря относительной индексации аргумент `cal` необходимо передавать только однократно, а не три раза.

Закрытие объекта Formatter

Как правило, после окончания эксплуатации объект Formatter должен быть закрыт, что освободит все ресурсы, которые он использовал. Это особенно важно при форматировании во время записи в файл, но может быть важно и в других случаях. В предшествующих примерах уже было продемонстрировано, что один из способов закрыть объект Formatter предусматривает явный вызов метода close(). Тем не менее, класс Formatter также реализует интерфейс AutoCloseable, т.е. поддерживает оператор try с ресурсами. При таком подходе объект Formatter автоматически закрывается, когда он больше не нужен.

В главе 13 был описан оператор try с ресурсами применительно к файлам, поскольку файлы являются одними из наиболее часто используемых ресурсов, которые необходимо закрывать. Однако основные приемы остаются неизменными. Например, ниже показан первый пример применения Formatter, переработанный для использования автоматического управления ресурсами:

```
// Использование автоматического управления ресурсами с классом Formatter
import java.util.*;

class FormatDemo {
  public static void main(String[] args) {

    try (Formatter fmt = new Formatter())
    {
      fmt.format("Форматировать %s легко: %d %f", "с помощью Java", 10, 98.6);
      System.out.println(fmt);
    }
  }
}
```

Вывод будет таким же, как и ранее.

Альтернативный вариант: метод printf()

Хотя формально нет ничего неправильного в использовании класса Formatter напрямую (как делалось в предыдущих примерах) при генерации выходных данных, которые будут отображаться на консоли, существует более удобная альтернатива: метод printf().

Метод printf() автоматически задействует Formatter для создания форматированной строки и затем отправляет эту строку в поток System.out, который по умолчанию является консолью. Метод printf() определен в классах PrintStream и PrintWriter и описан в главе 22.

Scanner

Класс Scanner является дополнением класса Formatter. Он обеспечивает чтение форматированных входных данных с последующим их преобразованием в двоичную форму. Класс Scanner можно применять для чтения входных данных из консоли, файла, строки или любого источника, который

реализует интерфейс Readable или ReadableByteChannel. Например, с использованием Scanner можно прочитать число с клавиатуры и затем присвоить его значение переменной. Вы увидите, что при всей его мощи класс Scanner на удивление прост в эксплуатации.

Конструкторы класса Scanner

В классе Scanner определено множество конструкторов, часть которых кратко описана в табл. 21.16.

Таблица 21.16. Избранные конструкторы класса Scanner

Конструктор	Описание
`Scanner(File from) throws FileNotFoundException`	Создает объект Scanner, который использует в качестве источника для ввода файл, указанный в from
`Scanner(File from, String charset) throws FileNotFoundException`	Создает объект Scanner, который использует в качестве источника для ввода файл, указанный в from, с кодировкой, заданной в charset
`Scanner(InputStream from)`	Создает объект Scanner, который использует в качестве источника для ввода поток данных, указанный в from
`Scanner(InputStream from, String charset)`	Создает объект Scanner, который использует в качестве источника для ввода поток данных, указанный в from, с кодировкой, заданной в charset
`Scanner(Path from) throws IOException`	Создает объект Scanner, который использует в качестве источника для ввода файл, указанный в from
`Scanner(Path from, String charset) throws IOException`	Создает объект Scanner, который использует в качестве источника для ввода файл, указанный в from, с кодировкой, заданной в charset
`Scanner(Readable from)`	Создает объект Scanner, который использует в качестве источника для ввода объект реализации Readable, указанный в from
`Scanner (ReadableByteChannel from)`	Создает объект Scanner, который использует в качестве источника для ввода объект реализации ReadableByteChannel, указанный в from

Окончание табл. 21.16

Конструктор	Описание
`Scanner (ReadableByteChannel from, String charset)`	Создает объект Scanner, который использует в качестве источника для ввода объект реализации ReadableByteChannel, указанный в from, с кодировкой, заданной в charset
`Scanner(String from)`	Создает объект Scanner, который использует в качестве источника для ввода строку, указанную в from

В общем случае объект Scanner может быть создан для String, InputStream, File, Path или любого объекта, который реализует интерфейс Readable или ReadableByteChannel. Рассмотрим несколько примеров.

Следующая кодовая последовательность создает объект Scanner, который читает из файла Test.txt:

```
FileReader fin = new FileReader("Test.txt");
Scanner src = new Scanner(fin);
```

Код работает, потому что класс FileReader реализует интерфейс Readable. Таким образом, вызов конструктора распознается как Scanner(Readable).

Показанная ниже строка создает объект Scanner, читающий из стандартного потока ввода, которым по умолчанию является клавиатура:

```
Scanner conin = new Scanner(System.in);
```

Код работает, поскольку System.in представляет собой объект типа InputStream. В результате вызов конструктора соответствует Scanner(InputStream).

Следующая кодовая последовательность создает объект Scanner, который читает из строки:

```
String instr = "10 99.88 сканировать легко.";
Scanner conin = new Scanner(instr);
```

Основы сканирования

После того, как объект Scanner создан, применять его для чтения форматированного ввода очень просто. В общем случае объект Scanner читает лексемы из базового источника, который был указан при его создании. Что касается Scanner, то лексема — это порция входных данных, ограниченная набором разделителей, которые по умолчанию являются пробельными символами. Лексема читается путем ее сопоставления с некоторым *регулярным выражением*, определяющим формат данных. Хотя класс Scanner позволяет определять конкретный тип выражения, с которым будет сопоставляться следующая операция ввода, он включает в себя множество предопределенных шаблонов, соответствующих примитивным типам, таким как int и double, и строкам. Таким образом, зачастую указывать шаблон для сопоставления не придется.

Как правило, для использования Scanner необходимо придерживаться описанной ниже процедуры.

1. Выяснить, доступен ли конкретный вид входных данных, вызвав один из методов hasNextX() класса Scanner, где X — желаемый тип данных.
2. Если входные данные доступны, тогда прочитать их, вызвав один из методов nextX() класса Scanner.
3. Повторять процесс до тех пор, пока входные данные не будут исчерпаны.
4. Закрыть объект Scanner, вызвав метод close().

Как следует из предыдущей процедуры, в классе Scanner определены два набора методов, позволяющих читать входные данные. В первый набор входят методы hasNextX(), кратко описанные в табл. 21.17, которые позволяют выяснить, доступен ли указанный тип входных данных. Скажем, метод hasNextInt() возвращает значение true, если следующая читаемая лексема является целым числом. Когда нужные данные доступны, их можно прочитать, вызывая один из методов nextX() класса Scanner, которые перечислены в табл. 21.18. Например, чтобы прочитать следующее целое число, необходимо вызвать метод nextInt(). В приведенной далее кодовой последовательности демонстрируется чтение списка целых чисел с клавиатуры.

```
Scanner conin = new Scanner(System.in);
int i;

// Прочитать список целых чисел.
while(conin.hasNextInt()) {
  i = conin.nextInt();
  // ...
}
```

Таблица 21.17. Методы hasNextX класса Scanner

Метод	Описание
boolean hasNext()	Возвращает true, если для чтения доступна еще одна лексема любого типа, или false в противном случае
boolean hasNext (Pattern pattern)	Возвращает true, если для чтения доступна лексема, которая соответствует шаблону, переданному в pattern, или false в противном случае
boolean hasNext (String pattern)	Возвращает true, если для чтения доступна лексема, которая соответствует шаблону, переданному в pattern, или false в противном случае
boolean hasNextBigDecimal()	Возвращает true, если для чтения доступно значение, которое может быть сохранено в объекте BigDecimal, или false в противном случае

Продолжение табл. 21.17

Метод	Описание
`boolean hasNextBigInteger()`	Возвращает `true`, если для чтения доступно значение, которое может быть сохранено в объекте `BigInteger`, или `false` в противном случае. Используется стандартное основание системы счисления (10, если оно не было изменено)
`boolean hasNextBigInteger (int radix)`	Возвращает `true`, если для чтения доступно значение с основанием системы счисления, указанным в `radix`, которое может быть сохранено в объекте `BigInteger`. В противном случае возвращается `false`
`boolean hasNextBoolean()`	Возвращает `true`, если для чтения доступно значение типа `boolean`, или `false` в противном случае
`boolean hasNextByte()`	Возвращает `true`, если для чтения доступно значение типа `byte`, или `false` в противном случае. Используется стандартное основание системы счисления (10, если оно не было изменено)
`boolean hasNextByte(int radix)`	Возвращает `true`, если для чтения доступно значение типа `byte` с основанием системы счисления, указанным в `radix`, или `false` в противном случае
`boolean hasNextDouble()`	Возвращает `true`, если для чтения доступно значение типа `double`, или `false` в противном случае
`boolean hasNextFloat()`	Возвращает `true`, если для чтения доступно значение типа `float`, или `false` в противном случае
`boolean hasNextInt()`	Возвращает `true`, если для чтения доступно значение типа `int`, или `false` в противном случае. Используется стандартное основание системы счисления (10, если оно не было изменено)
`boolean hasNextInt (int radix)`	Возвращает `true`, если для чтения доступно значение типа `int` с основанием системы счисления, указанным в `radix`, или `false` в противном случае
`boolean hasNextLine()`	Возвращает `true`, если для чтения доступна строчка входных данных
`boolean hasNextLong()`	Возвращает `true`, если для чтения доступно значение типа `long`, или `false` в противном случае. Используется стандартное основание системы счисления (10, если оно не было изменено)

Окончание табл. 21.17

Метод	Описание
`boolean hasNextLong (int radix)`	Возвращает `true`, если для чтения доступно значение типа `long` с основанием системы счисления, указанным в `radix`, или `false` в противном случае
`boolean hasNextShort()`	Возвращает `true`, если для чтения доступно значение типа `short`, или `false` в противном случае. Используется стандартное основание системы счисления (10, если оно не было изменено)
`boolean hasNextShort (int radix)`	Возвращает `true`, если для чтения доступно значение типа `short` с основанием системы счисления, указанным в `radix`, или `false` в противном случае

Таблица 21.18. Методы nextX класса Scanner

Метод	Описание
`String next()`	Возвращает из источника ввода следующую лексему любого типа
`String next (Pattern pattern)`	Возвращает из источника ввода следующую лексему, которая соответствует шаблону, переданному в `pattern`
`String next (String pattern)`	Возвращает из источника ввода следующую лексему, которая соответствует шаблону, переданному в `pattern`
`BigDecimal nextBigDecimal()`	Возвращает следующую лексему в виде объекта `BigDecimal`
`BigInteger nextBigInteger()`	Возвращает следующую лексему в виде объекта `BigInteger`. Используется стандартное основание системы счисления (10, если оно не было изменено)
`BigInteger nextBigInteger (int radix)`	Возвращает следующую лексему (с использованием указанного в `radix` основания системы счисления) в виде объекта `BigInteger`
`boolean nextBoolean()`	Возвращает следующую лексему в виде значения типа `boolean`
`byte nextByte()`	Возвращает следующую лексему в виде значения типа `byte`. Используется стандартное основание системы счисления (10, если оно не было изменено)

Окончание табл. 21.18

Метод	Описание
`byte nextByte (int radix)`	Возвращает следующую лексему (с использованием указанного в `radix` основания системы счисления) в виде значения типа `byte`
`double nextDouble()`	Возвращает следующую лексему в виде значения типа `double`
`float nextFloat()`	Возвращает следующую лексему в виде значения типа `float`
`int nextInt()`	Возвращает следующую лексему в виде значения типа `int`. Используется стандартное основание системы счисления (10, если оно не было изменено)
`int nextInt (int radix)`	Возвращает следующую лексему (с использованием указанного в `radix` основания системы счисления) в виде значения типа `int`
`String nextLine()`	Возвращает следующую строчку входных данных в виде объекта типа `String`
`long nextLong()`	Возвращает следующую лексему в виде значения типа `long`. Используется стандартное основание системы счисления (10, если оно не было изменено)
`long nextLong (int radix)`	Возвращает следующую лексему (с использованием указанного в `radix` основания системы счисления) в виде значения типа `long`
`short nextShort()`	Возвращает следующую лексему в виде значения типа `short`. Используется стандартное основание системы счисления (10, если оно не было изменено)
`short nextShort (int radix)`	Возвращает следующую лексему (с использованием указанного в `radix` основания системы счисления) в виде значения типа `short`

Цикл `while` останавливается, как только следующая лексема оказывается не целым числом. Таким образом, цикл прекращает чтение целых чисел, встретив в потоке ввода число, не являющееся целым.

Если методу `nextX()` не удается обнаружить данные искомого типа, тогда он генерирует исключение `InputMismatchException`. Когда доступных входных данных больше нет, генерируется исключение `NoSuchElementException`. По этой причине лучше всего сначала удостовериться в том, что данные желаемого типа доступны, вызвав метод `hasNextX()` перед вызовом соответствующего метода `nextX()`.

Примеры использования класса Scanner

Класс Scanner существенно облегчает решение того, что в противном случае было бы утомительной задачей. Чтобы понять почему, давайте рассмотрим несколько примеров. В следующей программе рассчитывается среднее значение для списка чисел, введенных с клавиатуры:

```
// Использование класса Scanner для расчета среднего по списку значений.
import java.util.*;

class AvgNums {
  public static void main(String[] args) {
    Scanner conin = new Scanner(System.in);

    int count = 0;
    double sum = 0.0;

    System.out.println("Введите числа для расчета среднего значения
(или done для завершения).");

    // Читать и суммировать числа.
    while(conin.hasNext()) {
      if(conin.hasNextDouble()) {
        sum += conin.nextDouble();
        count++;
      }
      else {
        String str = conin.next();
        if(str.equals("done")) break;
        else {
          System.out.println("Ошибка формата данных.");
          return;
        }
      }
    }

    conin.close();
    System.out.println("Среднее значение равно " + sum / count);
  }
}
```

В программе читаются числа с клавиатуры с суммированием их в процессе чтения, пока пользователь не введет строку done. Затем ввод останавливается и отображается среднее значение для введенных чисел. Ниже показан пример запуска:

```
Введите числа для расчета среднего значения (или done для завершения).
1.2
2
3.4
4
done
Среднее значение равно 2.65
```

Числа в программе читаются до тех пор, пока не встретится лексема, которая не представляет допустимое значение double. Когда такое происходит,

производится проверка, что лексема является строкой "done". Если это так, тогда программа завершается нормально, а иначе выдается сообщение об ошибке.

Обратите внимание, что числа читаются с помощью вызова nextDouble(). Метод nextDouble() читает любое число, которое может быть преобразовано в значение double, включая целочисленное значение наподобие 2 и значение с плавающей точкой вроде 3.4. Таким образом, в числе, прочитанном nextDouble(), десятичная точка совершенно не обязательна. Тот же общий принцип применим ко всем методам nextX(). Они сопоставляют и читают данные в любом формате, который может представлять тип запрашиваемого значения.

С классом Scanner связана одна примечательная особенность: прием, который используется для чтения из одного источника, можно применять для чтения из другого. Скажем, вот предыдущая программа, переработанная с целью расчета среднего значения для списка чисел, содержащихся в текстовом файле:

```
// Использование класса Scanner для расчета среднего по значениям в файле.
import java.util.*;
import java.io.*;

class AvgFile {
  public static void main(String[] args) throws IOException {

    int count = 0;
    double sum = 0.0;

    // Записать выходные данные в файл.
    FileWriter fout = new FileWriter("test.txt");
    fout.write("2 3.4 5 6 7.4 9.1 10.5 done");
    fout.close();

    FileReader fin = new FileReader("Test.txt");

    Scanner src = new Scanner(fin);

    // Читать и суммировать числа.
    while(src.hasNext()) {
      if(src.hasNextDouble()) {
        sum += src.nextDouble();
        count++;
      }
      else {
        String str = src.next();
        if(str.equals("done")) break;
        else {
          System.out.println("Ошибка формата файла.");
          return;
        }
      }
    }
    src.close();
    System.out.println("Среднее значение равно " + sum / count);
  }
}
```

Ниже представлен вывод:

```
Среднее значение равно 6.2
```

В предыдущей программе иллюстрируется еще одна важная особенность класса Scanner. Обратите внимание, что средство чтения файлов, на которое ссылается fin, не закрывается напрямую. Взамен оно закрывается автоматически, когда на src вызывается метод close(). При закрытии объекта Scanner ассоциированный с ним объект реализации Readable тоже закрывается (если данный объект реализации Readable реализует интерфейс Closeable). Следовательно, в таком случае файл, на который ссылается fin, автоматически закрывается при закрытии src.

Класс Scanner также реализует интерфейс AutoCloseable, т.е. им можно управлять с помощью блока try с ресурсами. Как объяснялось в главе 13, в случае использования try с ресурсами объект Scanner автоматически закрывается при окончании блока. Например, вот как можно было бы управлять src в предыдущей программе:

```
try (Scanner src = new Scanner(fin))
{
  // Читать и суммировать числа.
  while(src.hasNext()) {
    if(src.hasNextDouble()) {
      sum += src.nextDouble();
      count++;
    }
    else {
      String str = src.next();
      if(str.equals("done")) break;
      else {
        System.out.println("Ошибка формата файла.");
        return;
      }
    }
  }
}
```

Чтобы наглядно продемонстрировать закрытие объекта Scanner, в последующих примерах метод close() будет вызываться явно, но вы можете свободно применять оператор try с ресурсами в своем коде, где это уместно.

Еще один момент: ради компактности примеров в настоящем разделе исключения ввода-вывода просто генерируются в main(), но в реальном коде они обычно обрабатываются непосредственно.

Класс Scanner можно использовать для чтения входных данных разных типов, даже если их порядок не известен заранее. Необходимо лишь проверить, доступны ли данные того или иного типа, прежде чем читать их. Взгляните на следующую программу:

```
// Использование класса Scanner для чтения данных различных типов из файла.
import java.util.*;
import java.io.*;
```

```
class ScanMixed {
  public static void main(String[] args)
    throws IOException {

    int i;
    double d;
    boolean b;
    String str;

    // Записать выходные данные в файл.
    FileWriter fout = new FileWriter("test.txt");
    fout.write("Testing Scanner 10 12.2 one true two false");
    fout.close();

    FileReader fin = new FileReader("Test.txt");
    Scanner src = new Scanner(fin);

    // Читать до конца.
    while(src.hasNext()) {
      if(src.hasNextInt()) {
        i = src.nextInt();
        System.out.println("int: " + i);
      }
      else if(src.hasNextDouble()) {
        d = src.nextDouble();
        System.out.println("double: " + d);
      }
      else if(src.hasNextBoolean()) {
        b = src.nextBoolean();
        System.out.println("boolean: " + b);
      }
      else {
        str = src.next();
        System.out.println("String: " + str);
      }
    }
    src.close();
  }
}
```

Вот вывод:

```
String: Testing
String: Scanner
int: 10
double: 12.2
String: one
boolean: true
String: two
boolean: false
```

При чтении данных смешанных типов, как делается в предыдущей программе, важно внимательно следить за порядком, в котором вызываются методы `nextX()`. Например, если поменять местами в цикле вызовы `nextInt()` и `nextDouble()`, то оба числовых значения будут прочитаны как `double`, потому что для `nextDouble()` подходит любая строка, которая может быть представлена как число типа `double`.

Установка разделителей

Начало и окончание лексемы в классе Scanner определяется на основе набора *разделителей*. Стандартными разделителями являются пробельные символы, и именно они применялись в предшествующих примерах. Тем не менее, разделители можно изменить, вызвав метод useDelimiter():

```
Scanner useDelimiter(String pattern)
Scanner useDelimiter(Pattern pattern)
```

В pattern указывается регулярное выражение, которое задает набор разделителей. Ниже приведена переработанная версия показанной ранее программы расчета среднего значения, которая теперь читает список чисел, разделенных запятыми и любым количеством пробелов:

```
// Использование класса Scanner для расчета среднего по списку значений,
// разделенных запятыми.
import java.util.*;
import java.io.*;
class SetDelimiters {
  public static void main(String[] args)
    throws IOException {

    int count = 0;
    double sum = 0.0;
    // Записать выходные данные в файл.
    FileWriter fout = new FileWriter("test.txt");
    // Сохранить значения в списке, разделяя их запятыми.
    fout.write("2, 3.4,    5,6, 7.4, 9.1, 10.5, done");
    fout.close();

    FileReader fin = new FileReader("Test.txt");
    Scanner src = new Scanner(fin);

    // Установить пробел и запятую в качестве разделителей.
    src.useDelimiter(", *");

    // Читать и суммировать числа.
    while(src.hasNext()) {
      if(src.hasNextDouble()) {
        sum += src.nextDouble();
        count++;
      }
      else {
        String str = src.next();
        if(str.equals("done")) break;
        else {
          System.out.println("Ошибка формата файла.");
          return;
        }
      }
    }
    src.close();
    System.out.println("Среднее значение равно " + sum / count);
  }
}
```

В этой версии программы числа, записанные в файл test.txt, разделены запятыми и пробелами. Шаблон разделителей ", *" указывает объекту Scanner, что в качестве разделителей должны использоваться запятая и ноль или более пробелов. Вывод будет таким же, как и прежде.

Получить текущий шаблон разделителей позволяет метод delimiter():

```
Pattern delimiter()
```

Дополнительные средства класса Scanner

Помимо уже рассмотренных методов в классе Scanner определено несколько других. В некоторых случаях особенно полезным является метод findInLine():

```
String findInLine(Pattern pattern)
String findInLine(String pattern)
```

Метод findInLine() ищет указанный шаблон в следующей строчке текста. Если шаблон найден, тогда соответствующая ему лексема извлекается и возвращается. В противном случае возвращается null. Он работает независимо от набора разделителей и удобен, когда нужно отыскать специфический шаблон. Скажем, в следующей программе производится поиск во входной строке поля с возрастом (Age) и отображение возраста, если поле Age найдено:

```
// Демонстрация использования findInLine().
import java.util.*;
class FindInLineDemo {
  public static void main(String[] args) {
    String instr = "Name: Tom Age: 28 ID: 77";
    Scanner conin = new Scanner(instr);
    // Найти и отобразить возраст.
    conin.findInLine("Age:");  // найти Age
    if(conin.hasNext())
      System.out.println(conin.next());
    else
      System.out.println("Ошибка!");
    conin.close();
  }
}
```

Результатом будет 28. Метод findInLine() применяется в программе для поиска вхождения шаблона "Age". После обнаружения читается следующая лексема, которая и является возрастом.

С методом findInLine() связан метод findWithinHorizon():

```
String findWithinHorizon(Pattern pattern, int count)
String findWithinHorizon(String pattern, int count)
```

Метод findWithinHorizon() пытается найти вхождение указанного шаблона внутри следующих count символов. Если поиск успешен, тогда метод возвращает шаблон, давший совпадение, или null в противном случае. Когда значение count равно нулю, поиск выполняется по всем входным данным до тех пор, пока не будет найдено совпадение или не обнаружится конец входных данных.

Пропустить шаблон можно с помощью метода `skip()`:

```
Scanner skip(Pattern pattern)
Scanner skip(String pattern)
```

Если шаблон, указанный в `pattern`, найден, тогда `skip()` просто перемещается за его пределы и возвращает ссылку на вызывающий объект. Если шаблон не найден, то `skip()` генерирует исключение `NoSuchElementException`.

Другие методы класса `Scanner` включают `radix()`, который возвращает стандартное основание системы счисления, используемое в `Scanner`, `useRadix()`, устанавливающий основание системы счисления, `reset()`, который сбрасывает сканер, и `close()`, закрывающий сканер. В версии JDK 9 был добавлен метод `tokens()`, возвращающий все лексемы в виде `Stream<String>`, и метод `findAll()`, который возвращает лексемы, соответствующие указанному шаблону, в форме `Stream<MatchResult>`.

ResourceBundle, ListResourceBundle и PropertyResourceBundle

В состав пакета `java.util` входят три класса, которые помогают интернационализировать программы. Первым из них является абстрактный класс `ResourceBundle`. В нем определены методы, которые позволяют управлять коллекцией чувствительных к локали ресурсов, таких как строки, применяемые в качестве меток элементов пользовательского интерфейса в программе. Можно создать два и более набора строк, переведенных на разные языки, например, английский, немецкий или китайский, при этом каждый набор переведенных срок будет находиться в собственном комплекте. Затем понадобится загрузить комплект, соответствующий текущей локали, и использовать строки для построения пользовательского интерфейса программы.

Комплекты ресурсов идентифицируются по их *имени семейства* (также называемому *базовым именем*). К имени семейства может быть добавлен *код языка*, который определяет язык. В случае если запрошенная локаль соответствует коду языка, тогда применяется эта версия комплекта ресурсов. Например, комплект ресурсов с именем семейства `SampleRB` может иметь немецкую версию под названием `SampleRB_de` и французскую версию под названием `SampleRB_fr`. (Обратите внимание, что символ подчеркивания связывает имя семейства с кодом языка.) Таким образом, для локали `Locale.GERMAN` будет использоваться комплект ресурсов `SampleRB_de`.

Допускается также задавать конкретные варианты языка, относящиеся к определенной стране, указывая *код страны* после кода языка, скажем, `AU` для Австралии либо `IN` для Индии. Коду страны также предшествует символ подчеркивания, когда он связывается с именем комплекта ресурсов. Поддерживаются и другие вариации. Комплект ресурсов, который имеет только имя семейства, считается стандартным. Он применяется, когда нет подходящих комплектов, специфичных для языка.

На заметку! Коды языков определены в стандарте ISO 639, а коды стран — в стандарте ISO 3166.

Методы, определенные в классе ResourceBundle, кратко описаны в табл. 21.19. Важно отметить, что ключи null не разрешены и некоторые методы будут генерировать исключение NullPointerException, если в качестве ключа будет передано значение null. Обратите внимание на вложенный класс ResourceBundle.Control, который используется для управления процессом загрузки комплектов ресурсов.

Таблица 21.19. Методы, определенные в классе ResourceBundle

Метод	Описание
`static final void clearCache()`	Удаляет из кеша все комплекты ресурсов, которые были загружены загрузчиком классов. Начиная с версии JDK 9, данный метод удаляет из кеша все комплекты ресурсов, загруженные модулем, из которого вызывается этот метод
`static final void clearCache (ClassLoader ldr)`	Удаляет из кеша все комплекты ресурсов, которые были загружены ldr
`boolean containsKey (String k)`	Возвращает true, если k является ключом внутри вызывающего комплекта ресурсов (или его родителя)
`String getBaseBundleName()`	Возвращает базовое имя комплекта ресурсов, если оно доступно, или null в противном случае
`static final ResourceBundle getBundle (String familyName)`	Загружает комплект ресурсов с именем семейства familyName, используя стандартную локаль. Генерирует исключение MissingResourceException, если нет доступных комплектов ресурсов, соответствующих имени семейства
`static ResourceBundle getBundle (String familyName, Module mod)`	Загружает комплект ресурсов с именем семейства familyName для модуля, указанного в mod. Используется стандартная локаль. Генерирует исключение MissingResourceException, если нет доступных комплектов ресурсов, соответствующих имени семейства
`static final ResourceBundle getBundle (String familyName, Locale loc)`	Загружает комплект ресурсов с именем семейства familyName, используя указанную в loc локаль. Генерирует исключение MissingResourceException, если нет доступных комплектов ресурсов, соответствующих имени семейства

Продолжение табл. 21.19

Метод	Описание
`static ResourceBundle getBundle (String familyName, Locale loc, Module mod)`	Загружает комплект ресурсов с именем семейства `familyName`, используя указанную в `loc` локаль, для модуля, заданного в `mod`. Генерирует исключение `MissingResourceException`, если нет доступных комплектов ресурсов, соответствующих имени семейства
`static ResourceBundle getBundle (String familyName, Locale loc, ClassLoader ldr)`	Загружает комплект ресурсов с именем семейства `familyName`, используя указанную в `loc` локаль и заданный в `ldr` загрузчик классов. Генерирует исключение `MissingResourceException`, если нет доступных комплектов ресурсов, соответствующих имени семейства
`static final ResourceBundle getBundle(String familyName, ResourceBundle.Control cntl)`	Загружает комплект ресурсов с именем семейства `familyName`, используя стандартную локаль. Процесс загрузки управляется с помощью `cntl`. Генерирует исключение `MissingResourceException`, если нет доступных комплектов ресурсов, соответствующих имени семейства
`static final ResourceBundle getBundle (String familyName, Locale loc, ResourceBundle.Control cntl)`	Загружает комплект ресурсов с именем семейства `familyName`, используя указанную в `loc` локаль. Процесс загрузки управляется с помощью `cntl`. Генерирует исключение `MissingResourceException`, если нет доступных комплектов ресурсов, соответствующих имени семейства
`static ResourceBundle getBundle(String familyName, Locale loc, ClassLoader ldr, ResourceBundle.Control cntl)`	Загружает комплект ресурсов с именем семейства `familyName`, используя указанную в `loc` локаль и заданный в `ldr` загрузчик классов. Процесс загрузки управляется с помощью `cntl`. Генерирует исключение `MissingResourceException`, если нет доступных комплектов ресурсов, соответствующих имени семейства
`abstract Enumeration<String> getKeys()`	Возвращает ключи комплекта ресурсов в виде перечисления строк. Ключи родителя тоже извлекаются

Окончание табл. 21.19

Метод	Описание
`Locale getLocale()`	Возвращает локаль, поддерживаемую комплектом ресурсов
`final Object getObject(String k)`	Возвращает объект, который ассоциирован с ключом, переданным в k. Генерирует исключение `MissingResourceException`, если k отсутствует в комплекте ресурсов
`final String getString(String k)`	Возвращает строку, который ассоциирован с ключом, переданным в k. Генерирует исключение `MissingResourceException`, если k отсутствует в комплекте ресурсов. Генерирует исключение `ClassCastException`, если объект, ассоциированный с k, не является строкой
`final String[] getStringArray(String k)`	Возвращает строковый массив, который ассоциирован с ключом, переданным в k. Генерирует исключение `MissingResourceException`, если k отсутствует в комплекте ресурсов. Генерирует исключение `ClassCastException`, если объект, ассоциированный с k, не является строковым массивом
`protected abstract Object handleGetObject(String k)`	Возвращает объект, который ассоциирован с ключом, переданным в k. Возвращает `null`, если k отсутствует в комплекте ресурсов
`protected Set<String> handleKeySet()`	Возвращает ключи комплекта ресурсов в виде набора строк. Ключи родителя не извлекаются
`Set<String> keySet()`	Возвращает ключи комплекта ресурсов в виде набора строк. Ключи родителя тоже извлекаются
`protected void setParent (ResourceBundle parent)`	Устанавливает parent в качестве родительского комплекта для комплекта ресурсов. При поиске ключа будет выполняться поиск в родителе, если ключ не найден в вызывающем комплекте ресурсов

На заметку! Обратите внимание, что в версии JDK 9 в класс `ResourceBundle` добавлены методы, которые поддерживают модули. Кроме того, добавление модулей приводит к возникновению нескольких вопросов, связанных с использованием комплектов ресурсов, которые выходят за рамки настоящего обсуждения. Чтобы узнать, как модули влияют на работу с `ResourceBundle`, обратитесь в документацию по Java API.

Существуют два подкласса `ResourceBundle`. Первый — `PropertyResourceBundle`, который управляет ресурсами с помощью файлов свойств. Класс `PropertyResourceBundle` не добавляет собственных методов. Второй — абстрактный класс `ListResourceBundle`, управляющий ресурсами в массиве пар "ключ-значение". Класс `ListResourceBundle` добавляет метод `getContents()`, который должны реализовывать все подклассы:

```
protected abstract Object[][] getContents()
```

Метод `getContents()` возвращает двумерный массив, содержащий пары "ключ-значение", которые представляют ресурсы. Ключи обязаны быть строками. Значения обычно представляют собой строки, но могут быть и объектами других типов. Ниже приведен пример, демонстрирующий использование комплекта ресурсов в неименованном модуле. Комплект ресурсов имеет имя семейства `SampleRB`. Два класса комплектов ресурсов этого семейства создаются за счет расширения `ListResourceBundle`. Первый называется `SampleRB` и является стандартным комплектом, поддерживающим английский язык:

```
import java.util.*;
public class SampleRB extends ListResourceBundle {
  protected Object[][] getContents() {
    Object[][] resources = new Object[3][2];
    resources[0][0] = "title";
    resources[0][1] = "My Program";
    resources[1][0] = "StopText";
    resources[1][1] = "Stop";
    resources[2][0] = "StartText";
    resources[2][1] = "Start";
    return resources;
  }
}
```

Второй комплект ресурсов называется `SampleRB_de` и содержит перевод строк на немецкий язык:

```
import java.util.*;
// Версия для немецкого языка.
public class SampleRB_de extends ListResourceBundle {
  protected Object[][] getContents() {
    Object[][] resources = new Object[3][2];
    resources[0][0] = "title";
    resources[0][1] = "Mein Programm";
    resources[1][0] = "StopText";
    resources[1][1] = "Anschlag";
    resources[2][0] = "StartText";
    resources[2][1] = "Anfang";
    return resources;
  }
}
```

В следующей программе демонстрируется применение показанных выше двух комплектов ресурсов путем отображения строки, ассоциированной с каждым ключом, как для стандартной (английской) версии, так и для немецкой версии:

```
// Демонстрация использования комплектов ресурсов.
import java.util.*;

class LRBDemo {
  public static void main(String[] args) {
    // Загрузить стандартный комплект.
    ResourceBundle rd = ResourceBundle.getBundle("SampleRB");

    System.out.println("Английская версия: ");
    System.out.println("Строка для ключа title : " +
                       rd.getString("title"));

    System.out.println("Строка для ключа StopText: " +
                       rd.getString("StopText"));

    System.out.println("Строка для ключа StartText: " +
                       rd.getString("StartText"));

    // Загрузить комплект для немецкой версии.
    rd = ResourceBundle.getBundle("SampleRB", Locale.GERMAN);

    System.out.println("\nНемецкая версия: ");
    System.out.println("Строка для ключа title : " +
                       rd.getString("title"));

    System.out.println("Строка для ключа StopText: " +
                       rd.getString("StopText"));

    System.out.println("Строка для ключа StartText: " +
                       rd.getString("StartText"));
  }
}
```

Вот вывод:

```
Английская версия:
Строка для ключа title: My Program
Строка для ключа StopText: Stop
Строка для ключа StartText: Start

Немецкая версия:
Строка для ключа title: Mein Programm
Строка для ключа StopText: Anschlag
Строка для ключа StartText: Anfang
```

Смешанные служебные классы и интерфейсы

В дополнение к классам, которые уже обсуждались, пакет `java.util` содержит классы, кратко описанные в табл. 21.20.

Таблица 21.20. Дополнительные классы из пакета java.util

Класс	Описание
Base64	Поддерживает кодировку Base64. Определены также вложенные классы Encoder и Decoder
DoubleSummaryStatistics	Поддерживает сбор статистических данных для значений типа double. Доступны следующие статистические данные: среднее, минимум, максимум, подсчет и итог
EventListenerProxy	Расширяет класс EventListener, чтобы сделать возможной передачу дополнительных параметров. Прослушиватели событий обсуждаются в главе 25
EventObject	Суперкласс для всех классов событий. События обсуждаются в главе 25
FormattableFlags	Определяет флаги форматирования, используемые с интерфейсом Formattable
HexFormat	Обеспечивает разнообразные преобразования в и из шестнадцатеричных строк и цифр. Это класс, основанный на значении
IntSummaryStatistics	Поддерживает сбор статистических данных для значений типа int. Доступны следующие статистические данные: среднее, минимум, максимум, подсчет и итог
Objects	Различные методы, которые работают с объектами
PropertyPermission	Управляет разрешениями свойств
ServiceLoader	Предоставляет средства для поиска поставщиков служб
StringJoiner	Поддерживает конкатенацию объектов CharSequence, которые могут включать разделитель, префикс и суффикс
UUID	Инкапсулирует и управляет универсальными уникальными идентификаторами (Universally Unique Identifier — UUID)

Кроме того, в состав пакета java.util входят дополнительные интерфейсы, перечисленные в табл. 21.21.

Таблица 21.21. Дополнительные интерфейсы из пакета `java.util`

Интерфейс	Описание
`EventListener`	Указывает на то, что класс является прослушивателем событий. События обсуждаются в главе 25
`Formattable`	Позволяет классу обеспечивать специальное форматирование

Подпакеты пакета `java.util`

В Java определены следующие подпакеты пакета `java.util`:

- `java.util.concurrent`
- `java.util.concurrent.atomic`
- `java.util.concurrent.locks`
- `java.util.function`
- `java.util.jar`
- `java.util.logging`
- `java.util.prefs`
- `java.util.random`
- `java.util.regex`
- `java.util.spi`
- `java.util.stream`
- `java.util.zip`

Если не указано иное, то все они входят в состав модуля `java.base`. Каждый из них кратко рассматривается ниже.

`java.util.concurrent`, `java.util.concurrent.atomic` и `java.util.concurrent.locks`

Пакет `java.util.concurrent` вместе с двумя его подпакетами, `java.util.concurrent.atomic` и `java.util.concurrent.locks`, поддерживает параллельное программирование. Указанные пакеты предлагают высокопроизводительную альтернативу использованию встроенных средств синхронизации Java, когда требуются операции, безопасные в отношении потоков. Пакет `java.util.concurrent` также предоставляет инфраструктуру Fork/Join Framework. Эти пакеты подробно исследуются в главе 29.

`java.util.function`

В пакете `java.util.function` определено несколько функциональных интерфейсов, которые можно применять при создании лямбда-выражений или ссылок на методы. Они также широко используются в Java API.

Функциональные интерфейсы, определенные в `java.util.function`, показаны в табл. 21.22 вместе с кратким описанием их абстрактных методов. Имейте в виду, что в некоторых из этих интерфейсов также определены стандартные или статические методы, предоставляющие дополнительную функциональность. Обязательно выделите время на их самостоятельное изучение. (Применение функциональных интерфейсов обсуждалось в главе 15.)

Таблица 21.22. Функциональные интерфейсы, определенные в пакете `java.util.function`, и их абстрактные методы

Интерфейс	Абстрактный метод
`BiConsumer<T, U>`	`void accept(T tVal, U uVal)` Описание: воздействует на `tVal` и `uVal`
`BiFunction<T, U, R>`	`R apply(T tVal, U uVal)` Описание: воздействует на `tVal` и `uVal` и возвращает результат
`BinaryOperator<T>`	`T apply(T val1, T val2)` Описание: воздействует на два объекта того же самого типа и возвращает результат, который имеет такой же тип
`BiPredicate<T, U>`	`boolean test(T tVal, U uVal)` Описание: возвращает `true`, если `tVal` и `uVal` удовлетворяют условию, определенному с помощью `test()`, или `false` в противном случае
`BooleanSupplier`	`boolean getAsBoolean()` Описание: возвращает значение типа `boolean`
`Consumer<T>`	`void accept(T val)` Описание: воздействует на `val`
`DoubleBinaryOperator`	`double applyAsDouble(double val1, double val2)` Описание: воздействует на два значения типа `double` и возвращает результат типа `double`
`DoubleConsumer`	`void accept(double val)` Описание: воздействует на `val`
`DoubleFunction<R>`	`R apply(double val)` Описание: воздействует на значение типа `double` и возвращает результат

Продолжение табл. 21.22

Интерфейс	Абстрактный метод
DoublePredicate	boolean test(double val) Описание: возвращает true, если val удовлетворяет условию, определенному с помощью test(), или false в противном случае
DoubleSupplier	double getAsDouble() Описание: возвращает результат типа double
DoubleToIntFunction	int applyAsInt(double val) Описание: воздействует на значение типа double и возвращает результат типа int
DoubleToLongFunction	long applyAsLong(double val) Описание: воздействует на значение типа double и возвращает результат типа long
DoubleUnaryOperator	double applyAsDouble(double val) Описание: воздействует на значение типа double и возвращает результат типа double
Function<T, R>	R apply(T val) Описание: воздействует на val и возвращает результат
IntBinaryOperator	int applyAsInt(int val1, int val2) Описание: воздействует на два значения типа int и возвращает результат типа int
IntConsumer	int accept(int val) Описание: воздействует на val
IntFunction<R>	R apply(int val) Описание: воздействует на значение типа int и возвращает результат
IntPredicate	boolean test(int val) Описание: возвращает true, если val удовлетворяет условию, определенному с помощью test(), или false в противном случае
IntSupplier	int getAsInt() Описание: возвращает результат типа int
IntToDoubleFunction	double applyAsDouble(int val) Описание: воздействует на значение типа int и возвращает результат типа double

Продолжение табл. 21.22

Интерфейс	Абстрактный метод
IntToLongFunction	long applyAsLong(int val) Описание: воздействует на значение типа int и возвращает результат типа long
IntUnaryOperator	int applyAsInt(int val) Описание: воздействует на значение типа int и возвращает результат типа int
LongBinaryOperator	long applyAsLong(long val1, long val2) Описание: воздействует на два значения типа long и возвращает результат типа long
LongConsumer	void accept(long val) Описание: воздействует на val
LongFunction<R>	R apply(long val) Описание: воздействует на значение типа long и возвращает результат
LongPredicate	boolean test(long val) Описание: возвращает true, если val удовлетворяет условию, определенному с помощью test(), или false в противном случае
LongSupplier	long getAsLong() Описание: возвращает результат типа long
LongToDoubleFunction	double applyAsDouble(long val) Описание: воздействует на значение типа long и возвращает результат типа double
LongToIntFunction	int applyAsInt(long val) Описание: воздействует на значение типа long и возвращает результат типа int
LongUnaryOperator	long applyAsLong(long val) Описание: воздействует на значение типа long и возвращает результат типа long
ObjDoubleConsumer<T>	void accept(T val1, double val2) Описание: воздействует на значение val1 и значение val2 типа double
ObjIntConsumer<T>	void accept(T val1, int val2) Описание: воздействует на значение val1 и значение val2 типа int

Окончание табл. 21.22

Интерфейс	Абстрактный метод
ObjLongConsumer<T>	void accept(T val1, long val2) Описание: воздействует на значение val1 и значение val2 типа long
Predicate<T>	boolean test(T val) Описание: возвращает true, если val удовлетворяет условию, определенному с помощью test(), или false в противном случае
Supplier<T>	T get() Описание: возвращает объект типа T
ToDoubleBiFunction<T,U>	double applyAsDouble(T tVal, U uVal) Описание: воздействует на tVal и uVal и возвращает результат типа double
ToDoubleFunction<T>	double applyAsDouble(T val) Описание: воздействует на val и возвращает результат типа double
ToIntBiFunction<T,U>	int applyAsInt(T tVal, U uVal) Описание: воздействует на tVal и uVal и возвращает результат типа int
ToIntFunction<T>	int applyAsInt(T val) Описание: воздействует на val и возвращает результат типа int
ToLongBiFunction<T,U>	long applyAsLong(T tVal, U uVal) Описание: воздействует на tVal и uVal и возвращает результат типа long
ToLongFunction<T>	long applyAsLong(T val) Описание: воздействует на val и возвращает результат типа long
UnaryOperator<T>	T apply(T val) Описание: воздействует на val и возвращает результат

java.util.jar

Пакет java.util.jar предоставляет возможность чтения и записи файлов JAR (Java Archive — архив Java).

java.util.logging

Пакет java.util.logging обеспечивает поддержку журналов активности программ, которые можно использовать для записи действий программы, а также для поиска и устранения проблем. Этот пакет находится в модуле java.logging.

java.util.prefs

Пакет java.util.prefs обеспечивает поддержку пользовательских настроек. Обычно он применяется для поддержки конфигурации программ и находится в модуле java.prefs.

java.util.random

Пакет java.util.random предоставляет расширенную поддержку генераторов случайных чисел. (Добавлен в JDK 17.)

java.util.regex

Пакет java.util.regex обеспечивает поддержку обработки регулярных выражений. Он подробно описан в главе 31.

java.util.spi

Пакет java.util.spi поддерживает работу с поставщиками служб.

java.util.stream

Пакет java.util.stream содержит потоковый API для Java. Потоковый API обсуждается в главе 30.

java.util.zip

Пакет java.util.zip обеспечивает возможность чтения и записи файлов в популярных форматах ZIP и GZIP. Доступны потоки ввода и вывода ZIP и GZIP.

ГЛАВА

22 Ввод-вывод: исследование пакета java.io

В главе рассматривается пакет `java.io`, обеспечивающий поддержку операций ввода-вывода. В главе 13 был представлен обзор системы ввода-вывода Java — основные приемы чтения и записи файлов, обработка исключений ввода-вывода и закрытие файлов. Здесь мы исследуем систему ввода-вывода Java более подробно.

Как уже давно известно всем программистам, большинство программ не могут достичь своих целей без доступа к внешним данным. Данные извлекаются из источника *ввода*. Результаты программы отправляются в место назначения *вывода*. В Java такие источники или места назначения определяются очень широко. Например, классы ввода-вывода Java способны управлять сетевым подключением, буфером памяти или дисковым файлом. Будучи физически разными, все эти устройства управляются одной и той же абстракцией: *потоком*. Как объяснялось в главе 13, поток ввода-вывода представляет собой логическую сущность, которая либо выпускает, либо потребляет информацию. Поток ввода-вывода связан с физическим устройством через систему ввода-вывода Java. Все потоки ввода-вывода ведут себя одинаково, даже если реальные физические устройства, с которыми они связаны, различаются.

На заметку! Основанная на потоках система ввода-вывода из пакета `java.io`, рассматриваемая в этой главе, была частью языка Java с момента его первого выпуска и применяется крайне широко. Однако, начиная с версии 1.4, в Java появилась вторая система ввода-вывода, которая называется NIO (первоначально аббревиатура от "New I/O"). Система NIO находится в пакете `java.nio` и его подпакетах и NIO будет описана в главе 23.

На заметку! Важно не путать потоки ввода-вывода, используемые в обсуждаемой здесь системе ввода-вывода, с потоковым API, добавленным в версии JDK 8. Они являются концептуально связанными, но разными сущностями. Таким образом, когда в главе встречается термин "поток", он относится к потоку ввода-вывода.

Классы и интерфейсы ввода-вывода

Ниже перечислены классы ввода-вывода, определенные в пакете `java.io`:

`BufferedInputStream`	`FileWriter`	`PipedInputStream`
`BufferedOutputStream`	`FilterInputStream`	`PipedOutputStream`
`BufferedReader`	`FilterOutputStream`	`PipedReader`
`BufferedWriter`	`FilterReader`	`PipedWriter`
`ByteArrayInputStream`	`FilterWriter`	`PrintStream`
`ByteArrayOutputStream`	`InputStream`	`PrintWriter`
`CharArrayReader`	`InputStreamReader`	`PushbackInputStream`
`CharArrayWriter`	`LineNumberReader`	`PushbackReader`
`Console`	`ObjectInputFilter.Config`	`RandomAccessFile`
`DataInputStream`	`ObjectInputStream`	`Reader`
`DataOutputStream`	`ObjectInputStream.GetField`	`SequenceInputStream`
`File`	`ObjectOutputStream`	`SerializablePermission`
`FileDescriptor`	`ObjectOutputStream.PutField`	`StreamTokenizer`
`FileInputStream`	`ObjectStreamClass`	`StringReader`
`FileOutputStream`	`ObjectStreamField`	`StringWriter`
`FilePermission`	`OutputStream`	`Writer`
`FileReader`	`OutputStreamWriter`	

Пакет `java.io` содержит также два нерекомендуемых класса, отсутствующих в предыдущем списке: `LineNumberInputStream` и `StringBufferInputStream`. В новом коде они применяться не должны.

В `java.io` определены следующие интерфейсы:

`Closeable`	`FilenameFilter`	`ObjectInputValidation`
`DataInput`	`Flushable`	`ObjectOutput`
`DataOutput`	`ObjectInput`	`ObjectStreamConstants`
`Externalizable`	`ObjectInputFilter`	`Serializable`
`FileFilter`	`ObjectInputFilter.FilterInfo`	

Как видите, в пакете `java.io` есть много классов и интерфейсов. К ним относятся байтовые и символьные потоки, а также сериализация объектов (хранение и извлечение объектов). В настоящей главе рассматривается ряд часто используемых компонентов ввода-вывода. Обсуждение начинается с одного из наиболее характерных классов ввода-вывода: `File`.

File

Хотя большинство классов, определенных в пакете `java.io`, работают с потоками, это не касается класса `File`, который имеет дело напрямую с файлами и файловой системой. То есть класс `File` не определяет, каким обра-

зом информация извлекается из файлов или сохраняется в них; он описывает свойства самого файла. Объект `File` применяется для получения или управления информацией, связанной с файлом на диске, такой как разрешения, время, дата и путь к каталогу, а также для навигации по иерархии подкаталогов.

На заметку! Интерфейс `Path` и класс `Files`, входящие в состав системы NIO, во многих случаях предлагают мощную альтернативу классу `File`. Детальные сведения ищите в главе 23.

Файлы являются основным источником и местом назначения данных во многих программах. Хотя из соображений безопасности существуют серьезные ограничения на их использование в не заслуживающем доверия коде, файлы по-прежнему являются центральным ресурсом для хранения постоянной и общей информации. Каталог в Java рассматривается просто как файл с одним дополнительным свойством — списком имен файлов, которые можно просмотреть с помощью метода `list()`.

Для создания объектов `File` предназначены следующие конструкторы:

```
File(String directoryPath)
File(String directoryPath, String filename)
File(File dirObj, String filename)
File(URI uriObj)
```

Здесь в `directoryPath` указывается имя пути к каталогу, в `filename` — имя файла или подкаталога, в `dirObj` — объект `File`, представляющий каталог, а в `uriObj` — объект `URI`, описывающий файл.

В показанном ниже примере создаются три файла: `f1`, `f2` и `f3`. Первый объект `File` создается с путем к каталогу в качестве единственного аргумента. Второй объект `File` создается с указанием двух аргументов — пути и имени файла. Третий объект `File` создается с применением пути к файлу, назначенному `f1`, и имени файла; `f3` ссылается на тот же файл, что и `f2`.

```
File f1 = new File("/");
File f2 = new File("/","autoexec.bat");
File f3 = new File(f1,"autoexec.bat");
```

На заметку! Компилятор Java правильно интерпретирует разделители путей в соответствии с соглашениями UNIX и Windows. В случае использования косой черты (/) в версии Java для Windows путь все равно будет распознан корректно. Не забывайте, что если применяется соглашение Windows о символе обратной косой черты (\), то внутри строки придется использовать соответствующую управляющую последовательность (\\).

В классе `File` определено множество методов, которые получают стандартные свойства объекта `File`. Скажем, `getName()` возвращает имя файла, `getParent()` возвращает имя родительского каталога, а `exists()` возвращает `true`, если файл существует, или `false`, если нет. В приведенном далее примере демонстрируется применение нескольких методов класса `File`. Предполагается, что в корневом каталоге присутствует каталог с именем `java`, содержащий файл по имени `COPYRIGHT`.

```
// Демонстрация методов класса File.
import java.io.File;

class FileDemo {
  static void p(String s) {
    System.out.println(s);
  }

  public static void main(String[] args) {
    File f1 = new File("/java/COPYRIGHT");

    p("Имя файла: " + f1.getName());
    p("Путь: " + f1.getPath());
    p("Абсолютный путь: " + f1.getAbsolutePath());
    p("Родительский каталог: " + f1.getParent());
    p(f1.exists() ? "существует" : "не существует");
    p(f1.canWrite() ? "допускает запись" : "не допускает запись");
    p(f1.canRead() ? "допускает чтение" : "не допускает чтение");
    p(f1.isDirectory() ? "" : "не " + "является каталогом");
    p(f1.isFile() ? "является нормальным файлом" : "может быть именованным каналом");
    p(f1.isAbsolute() ? "является абсолютным" : "не является абсолютным");
    p("Файл изменялся в последний раз: " + f1.lastModified());
    p("Размер файла: " + f1.length() + " байт(ов)");
  }
}
```

Программа сгенерирует вывод следующего вида:

```
Имя файла: COPYRIGHT
Путь: \java\COPYRIGHT
Абсолютный путь: C:\java\COPYRIGHT
Родительский каталог: \java
существует
допускает запись
допускает чтение
не является каталогом
является нормальным файлом
не является абсолютным
Файл изменялся в последний раз: 1282832030047
Размер файла: 695 байт(ов)
```

Большинство методов класса `File` самоочевидно, но ситуация с методами `isFile()` и `isAbsolute()` иная. Метод `isFile()` возвращает `true`, если вызывается для файла, или `false` — если для каталога. Кроме того, `isFile()` возвращает `false` для ряда специальных файлов, таких как драйверы устройств и именованные каналы, поэтому данный метод можно использовать, чтобы удостовериться в том, что файл ведет себя как файл. Метод `isAbsolute()` возвращает `true`, если файл имеет абсолютный путь, или `false`, если его путь является относительным.

В классе `File` есть два полезных служебных метода, представляющих особый интерес. Первый из них — `renameTo()`:

```
boolean renameTo(File newName)
```

Имя файла, заданное параметром newName, становится новым именем файла. Метод возвращает true в случае успеха или false, если файл не может быть переименован (например, при попытке переименования файла с указанием существующего имени).

Второй служебный метод, delete(), удаляет дисковый файл, представленный путем вызывающего объекта File:

```
boolean delete()
```

Метод delete() можно также применять для удаления каталога, если он пуст. Метод delete() возвращает true, если файл был удален, или false, если удалить файл невозможно. В табл. 22.1 кратко описаны другие методы класса File, которые вы можете счесть полезными.

Таблица 22.1. Полезные методы класса File

Метод	Описание
`void deleteOnExit()`	Удаляет файл, ассоциированный с вызывающим объектом, когда виртуальная машина Java прекращает работу
`long getFreeSpace()`	Возвращает количество свободных байтов, которые доступны в разделе, ассоциированном с вызывающим объектом
`long getTotalSpace()`	Возвращает емкость раздела, ассоциированного с вызывающим объектом
`long getUsableSpace()`	Возвращает количество пригодных для потребления свободных байтов, которые доступны в разделе, ассоциированном с вызывающим объектом
`boolean isHidden()`	Возвращает true, если вызывающий файл является скрытым, или false в противном случае
`boolean setLastModified (long millisec)`	Устанавливает отметку времени для вызывающего файла в значение millisec, которое представляет собой количество миллисекунд, прошедших с 1 января 1970 года, UTC
`boolean setReadOnly()`	Делает вызывающий файл доступным только для чтения

Кроме того, существуют методы, позволяющие помечать файлы как допускающие чтение, запись и выполнение. Поскольку класс File реализует интерфейс Comparable, также поддерживается метод compareTo().

Особый интерес вызывает метод `toPath()`:

```
Path toPath()
```

Метод `toPath()` возвращает объект `Path`, который представляет файл, инкапсулированный вызывающим объектом `File`. (Другими словами, `toPath()` преобразует объект `File` в объект `Path`.) Класс `Path` находится в пакете `java.nio.file` и является частью системы NIO. Таким образом, метод `toPath()` образует мост между старым классом `File` и новым интерфейсом `Path`. (Класс `Path` обсуждается в главе 23.)

Каталоги

Каталог представляет собой объект `File`, содержащий список других файлов и каталогов. Для созданного объекта `File`, который является каталогом, метод `isDirectory()` возвращает значение `true`. В таком случае можно вызвать метод `list()` на этом объекте для извлечения списка других файлов и каталогов внутри. Метод `list()` имеет две формы, первая из которых показана ниже:

```
String[] list()
```

Список файлов возвращается в массиве объектов `String`.

В приведенной далее программе демонстрируется использование метода `list()` для исследования содержимого каталога:

```
// Демонстрация работы с каталогами.
import java.io.File;

class DirList {
  public static void main(String[] args) {
    String dirname = "/java";
    File f1 = new File(dirname);

    if(f1.isDirectory()) {
      System.out.println("Каталог " + dirname);
      String[] s = f1.list();

      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " - каталог");
        } else {
          System.out.println(s[i] + " - файл");
        }
      }
    } else {
      System.out.println(dirname + " не является каталогом");
    }
  }
}
```

Ниже показан пример вывода, сгенерированного программой. (В зависимости от каталога вывод будет отличаться.)

```
Каталог /java
bin - каталог
lib - каталог
demo - каталог
COPYRIGHT - файл
README - файл
index.html - файл
include - каталог
src.zip - файл
src - каталог
```

Использование интерфейса `FilenameFilter`

Часто желательно ограничить количество файлов, возвращаемых методом `list()`, чтобы включать только те файлы, которые соответствуют определенному шаблону имен файлов, или *фильтру*. Для этого должна применяться вторая форма метода `list()`:

```
String[] list(FilenameFilter FFObj)
```

В `FFObj` передается объект класса, реализующего интерфейс `FilenameFilter`.

В интерфейсе `FilenameFilter` определен единственный метод `accept()`, который вызывается по одному разу для каждого файла в списке. Ниже приведен его общий вид:

```
boolean accept(File directory, String filename)
```

Метод `accept()` возвращает `true` для файлов в каталоге, указанном в `directory`, которые должны быть включены в список (т.е. те, что соответствуют аргументу `filename`), или `false` для файлов, которые должны быть исключены из списка.

Показанный далее класс `OnlyExt` реализует интерфейс `FilenameFilter`. Он будет использоваться для изменения предыдущей программы, чтобы ограничить видимость имен файлов, возвращаемых `list()`, файлами с именами, которые заканчиваются расширением, указанным при конструировании объекта.

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
  String ext;

  public OnlyExt(String ext) {
    this.ext = "." + ext;
  }

  public boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}
```

Вот модифицированная программа для получения листинга каталогов. Теперь она будет отображать только файлы с расширением `.html`.

```
// Вывод списка файлов .HTML в каталоге.
import java.io.*;

class DirListOnly {
  public static void main(String[] args) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new OnlyExt("html");
    String[] s = f1.list(only);

    for (int i=0; i < s.length; i++) {
      System.out.println(s[i]);
    }
  }
}
```

Альтернативные методы `listFiles()`

Может оказаться полезной разновидность метода `list()`, называемая `listFiles()`. Ниже перечислены сигнатуры `listFiles()`:

```
File[] listFiles()
File[] listFiles(FilenameFilter FFObj)
File[] listFiles(FileFilter FObj)
```

Методы `listFiles()` возвращают список файлов в виде массива объектов `File`, а не строк. Первый метод возвращает все файлы, а второй — те файлы, которые удовлетворяют указанному фильтру `FilenameFilter`. Помимо возвращения массива объектов `File` первые две формы `listFiles()` работают подобно эквивалентным методам `list()`.

Третья форма `listFiles()` возвращает файлы с именами путей, которые удовлетворяют указанному объекту `FileFilter`. В классе `FileFilter` определен единственный метод `accept()`, вызываемый по одному разу для каждого файла в списке. Вот его общий вид:

```
boolean accept(File path)
```

Метод `accept()` возвращает `true` для файлов, которые должны быть включены в список (т.е. соответствующих аргументу `path`), или `false` для файлов, которые должны быть исключены из списка.

Создание каталогов

В классе `File` есть еще два полезных служебных метода — `mkdir()` и `mkdirs()`. Метод `mkdir()` создает каталог, возвращая `true` в случае успеха или `false` в случае неудачи. Отказ может произойти по разным причинам, например, если путь, указанный в объекте `File`, уже существует, или каталог невозможно создать, т.к. полный путь пока не существует. Создать каталог, для которого отсутствует полный путь, позволяет метод `mkdirs()`. Он создает каталог и всех его родителей.

Интерфейсы AutoCloseable, Closeable и Flushable

Для потоковых классов очень важны три интерфейса: Closeable и Flushable, которые определены в java.io, и AutoCloseable, упакованный в java.lang.

Интерфейс AutoCloseable обеспечивает поддержку для оператора try с ресурсами, который автоматизирует процесс закрытия ресурса (см. главу 13). Управлять с помощью оператора try с ресурсами можно только объектами классов, реализующих AutoCloseable. Интерфейс AutoCloseable обсуждался в главе 18, а здесь он упоминается ради удобства. В нем определен единственный метод close():

```
void close() throws Exception
```

Метод close() закрывает вызывающий объект, освобождая все ресурсы, которые он может удерживать. Он вызывается автоматически в конце оператора try с ресурсами, что устраняет необходимость в явном вызове close(). Поскольку интерфейс AutoCloseable реализуется всеми классами ввода-вывода, которые открывают поток данных, все такие потоки могут быть автоматически закрыты посредством оператора try с ресурсами. Автоматическое закрытие потока гарантирует, что он будет корректно закрыт, когда больше не нужен, что предотвращает утечку памяти и другие проблемы.

В интерфейсе Closeable тоже определен метод close(). Объекты класса, реализующие Closeable, могут быть закрыты. Интерфейс Closeable расширяет AutoCloseable. Следовательно, любой класс, реализующий Closeable, реализует и AutoCloseable.

Объекты класса, реализующего Flushable, могут принудительно записывать буферизованный вывод в поток данных, к которому присоединен объект. В интерфейсе Flushable определен метод flush():

```
void flush() throws IOException
```

Очистка потока данных обычно приводит к физической записи буферизованного вывода на базовое устройство. Интерфейс Flushable реализуется всеми классами ввода-вывода, которые выполняют запись в поток данных.

Исключения ввода-вывода

В обработке ввода-вывода важную роль играют два исключения. Первое исключение — IOException. Что касается большинства классов ввода-вывода, описанных в главе, то при возникновении ошибки ввода-вывода генерируется исключение IOException. Во многих случаях, когда файл не может быть открыт, генерируется исключение FileNotFoundException. Класс FileNotFoundException является подклассом IOException, поэтому они оба могут быть перехвачены с помощью одного оператора catch, который перехватывает IOException. Для краткости такой подход используется в

большинстве примеров кода в главе. Тем не менее, в реальных приложениях часто полезно перехватывать каждое исключение по отдельности.

Временами при выполнении операций ввода-вывода важен еще один класс исключений — `SecurityException`. Как объяснялось в главе 13, в ситуациях, когда присутствует диспетчер безопасности, несколько классов, представляющих файлы, будут генерировать исключение `SecurityException`, если при попытке открытия файла произойдет нарушение ограничений безопасности. По умолчанию приложения, запускаемые через `java`, не задействуют диспетчер безопасности. По этой причине в примерах ввода-вывода, рассматриваемых в книге, нет необходимости отслеживать возможное исключение `SecurityException`. Однако другие приложения могут генерировать исключение `SecurityException`, и тогда придется его обработать. Но имейте в виду, что в JDK 17 диспетчер безопасности объявлен устаревшим и подлежащим удалению.

Два способа закрытия потока данных

Как правило, когда поток больше не нужен, он должен быть закрыт. Невыполнение такого требования может привести к утечке памяти и нехватке ресурсов. Приемы, применяемые для закрытия потока, были описаны в главе 13, но из-за их важности они заслуживают краткого повторения перед тем, как приступать к исследованию классов потоков.

Существуют два основных способа закрытия потока данных. Первый способ предусматривает явный вызов метода `close()` на потоке. Это традиционный подход, который использовался с момента первоначального выпуска Java. При таком подходе `close()` обычно вызывается внутри блока `finally`. Таким образом, вот упрощенная структура, отражающая традиционный подход:

```
try {
  // открыть файл и работать с ним
} catch(искл-ввода-вывода) {
  // ...
} finally {
  // закрыть файл
}
```

Здесь *искл-ввода-вывода* — исключение ввода-вывода. Показанный общий прием (или его вариант) был распространен в коде, предшествующем выходу JDK 7. Второй подход к закрытию потока заключается в автоматизации процесса с помощью оператора `try` с ресурсами, добавленного в JDK 7. Оператор `try` с ресурсами является расширенной формой `try`, имеющей следующий вид:

```
try (спецификация-ресурса) {
  // использовать ресурс
}
```

Обычно в спецификации ресурса указывается оператор или операторы, которые объявляют и инициализируют ресурс, такой как файл или другой

ресурс, связанный с потоком. Он состоит из объявления переменной, в котором переменная инициализируется ссылкой на управляемый объект. Когда блок `try` заканчивается, ресурс автоматически освобождается. В случае файла это означает, что файл автоматически закрывается. Таким образом, нет необходимости вызывать метод `close()` явно. Начиная с JDK 9, спецификация ресурса в `try` также может состоять из переменной, которая была объявлена и инициализирована ранее в программе. Однако такая переменная обязана быть фактически финальной, т.е. после присваивания начального значения новое значение ей присваиваться не должно. Ниже перечислены три ключевых момента, касающихся оператора `try` с ресурсами.

- Ресурсы, управляемые оператором `try` с ресурсами, должны быть объектами классов, которые реализуют интерфейс `AutoCloseable`.
- Ресурс, объявленный в `try`, является неявно финальным. Ресурс, объявленный вне `try`, должен быть фактически финальным.
- Управлять можно более чем одним ресурсом, разделяя каждое объявление точкой с запятой.

Кроме того, не забывайте о том, что область видимости ресурса, объявленного внутри `try`, ограничена оператором `try` с ресурсами.

Главное преимущество оператора `try` с ресурсами связано с тем, что ресурс (в данном случае поток данных) автоматически закрывается, когда заканчивается блок `try`. Таким образом, например, невозможно забыть о закрытии потока. Подход с оператором `try` с ресурсами также обычно приводит к более короткому, ясному и легкому в сопровождении исходному коду.

Ожидается, что благодаря своим преимуществам оператор `try` с ресурсами будет широко применяться в новом коде, а потому он будет использоваться в большей части кода в этой главе (и повсюду в книге). Тем не менее, поскольку по-прежнему существует большой объем старого кода, всем программистам важно также ознакомиться с традиционным подходом к закрытию потока данных. Например, по всей видимости, вам придется работать с унаследованным кодом, где применяется традиционный подход, или в среде, использующей более старую версию Java. Также могут возникать ситуации, когда автоматизированный подход непригоден из-за других аспектов вашего кода. По этой причине в нескольких примерах ввода-вывода в книге будет демонстрироваться традиционный подход, чтобы вы могли увидеть его в действии.

И последнее замечание: примеры, в которых применяется оператор `try` с ресурсами, должны быть скомпилированы компилятором Java современной версии. Со старым компилятором они работать не будут. Примеры, использующие традиционный подход, могут компилироваться компилятором Java более старой версии.

Помните! Из-за того, что оператор `try` с ресурсами упрощает процесс освобождения ресурса и устраняет возможность случайного забывания освободить ресурс, такой подход рекомендуется применять в новом коде, когда это уместно.

Классы потоков данных

Потоковый ввод-вывод в Java основан на четырех абстрактных классах: `InputStream`, `OutputStream`, `Reader` и `Writer`. Упомянутые классы были кратко рассмотрены в главе 13. Они используются для создания нескольких конкретных подклассов потоков. Хотя ваши программы выполняют свои операции ввода-вывода через конкретные подклассы, классы верхнего уровня определяют базовую функциональность, общую для всех потоковых классов.

Классы `InputStream` и `OutputStream` предназначены для байтовых потоков, а классы `Reader` и `Writer` — для символьных потоков. Классы байтовых потоков и классы символьных потоков образуют отдельные иерархии. В общем случае классы символьных потоков должны применяться при работе с символами или строками, тогда как классы байтовых потоков — при работе с байтами или другими двоичными объектами. В оставшемся материале главы исследуются как байтовые, так и символьные потоки.

Байтовые потоки

Классы байтовых потоков предоставляют развитую среду для обработки ввода-вывода, ориентированного на байты. Байтовый поток можно использовать с любым типом объекта, включая двоичные данные. Универсальность подобного рода делает байтовые потоки важными для многих типов программ. Поскольку на верхушке иерархии классов байтовых потоков находятся `InputStream` и `OutputStream`, с них и начинается обсуждение.

InputStream

`InputStream` — это абстрактный класс, определяющий модель потокового байтового ввода в Java. Он реализует интерфейсы `AutoCloseable` и `Closeable`. Большинство методов класса `InputStream` (кроме `mark()` и `markSupported()`) при возникновении ошибки ввода-вывода будут генерировать исключение `IOException`. В табл. 22.2 кратко описаны методы класса `InputStream`.

Таблица 22.2. Методы, определенные в классе `InputStream`

Метод	Описание
`int available()`	Возвращает количество байтов входных данных, доступных в текущий момент для чтения
`void close()`	Закрывает источник ввода. Дальнейшие попытки чтения приведут к генерации исключения `IOException`
`void mark(int numBytes)`	Помещает метку в текущую позицию потока ввода, которая будет оставаться действительной до тех пор, пока не будет прочитано `numBytes` байтов

Продолжение табл. 22.2

Метод	Описание
`boolean markSupported()`	Возвращает `true`, вызывающий поток поддерживает методы `mark()/reset()`
`static InputStream nullInputStream()`	Возвращает открытый, но пустой поток ввода, т.е. поток, не содержащий данных. Таким образом, текущая позиция всегда находится в конце потока, и никакие входные данные не могут быть получены. Однако поток можно закрыть
`int read()`	Возвращает целочисленное представление следующего доступного байта во входных данных. При попытке чтения в конце потока возвращает `-1`
`int read(byte[] buffer)`	Пытается прочитать вплоть до `buffer.length` байтов в буфер, указанный в `buffer`, и возвращает фактическое количество успешно прочитанных байтов. При попытке чтения в конце потока возвращает `-1`
`int read(byte[] buffer, int offset, int numBytes)`	Пытается прочитать вплоть до `numBytes` байтов в буфер `buffer`, начиная с `buffer[offset]`, и возвращает фактическое количество успешно прочитанных байтов. При попытке чтения в конце потока возвращает `-1`
`byte[] readAllBytes()`	Начиная с текущей позиции, читает до конца потока и возвращает байтовый массив, который содержит входные данные
`byte[] readNBytes (int numBytes)`	Пытается прочитать `numBytes` байтов и возвращает результат в байтовом массиве. Если конец потока достигнут до того, как было прочитано `numBytes` байтов, тогда возвращаемый массив будет содержать меньше, чем `numBytes` байтов
`int readNBytes(byte[] buffer, int offset, int numBytes)`	Пытается прочитать вплоть до `numBytes` байтов в буфер `buffer`, начиная с `buffer[offset]`, и возвращает фактическое количество успешно прочитанных байтов
`void reset()`	Переустанавливает указатель ввода на ранее установленную метку
`long skip (long numBytes)`	Игнорирует (т.е. пропускает) `numBytes` байтов входных данных и возвращает фактическое количество пропущенных байтов

Окончание табл. 22.2

Метод	Описание
`void skipNBytes (long numBytes)`	Игнорирует (т.е. пропускает) `numBytes` байтов входных данных. Генерирует исключение `EOFException`, если достигнут конец потока до того, как было пропущено `numBytes` байтов, или генерирует исключение `IOException` при возникновении какой-то ошибки ввода-вывода
`long transferTo (OutputStream strm)`	Копирует байты из вызывающего потока в `strm` и возвращает количество скопированных байтов

На заметку! Большинство методов, описанных в табл. 22.2, реализовано в подклассах `InputStream` (кроме `mark()` и `reset()`); в последующем обсуждении обращайте внимание на их применение или отсутствие в каждом подклассе.

OutputStream

`OutputStream` — абстрактный класс, определяющий потоковый вывод байтов. Он реализует интерфейсы `AutoCloseable`, `Closeable` и `Flushable`. При возникновении ошибок ввода-вывода большинство методов, определенных в этом классе, возвращают значение `void` и генерируют исключение `IOException`. Методы класса `OutputStream` кратко описаны в табл. 22.3.

Таблица 22.3. Методы, определенные в классе `OutputStream`

Метод	Описание
`void close()`	Закрывает поток вывода. Дальнейшие попытки записи приведут к генерации исключения `IOException`
`void flush()`	Финализирует состояние вывода, так что любые буферы очищаются, т.е. сбрасывает буферы вывода
`static OutputStream nullOutputStream()`	Возвращает открытый, но пустой поток вывода, т.е. поток, в который никакие выходные данные фактически не записывались. Таким образом, методы вывода могут быть вызваны, но они не производят какой-либо вывод. Однако поток можно закрыть
`void write(int b)`	Записывает одиночный байт в поток вывода. Обратите внимание, что параметр имеет тип `int`, что позволяет вызывать `write()` с выражением без необходимости в приведении к типу `byte`

Окончание табл. 22.3

Метод	Описание
`void write(byte[] buffer)`	Записывает полный байтовый массив в поток вывода
`void write(byte[] buffer, int offset, int numBytes)`	Записывает поддиапазон длиной numBytes байтов из буфера buffer типа байтового массива, начиная с `buffer[offset]`

FileInputStream

Класс `FileInputStream` создает объект `InputStream`, который можно применять для чтения байтов из файла. Ниже показаны два часто используемых конструктора:

```
FileInputStream(String filePath)
FileInputStream(File fileObj)
```

Любой из них может сгенерировать исключение `FileNotFoundException`. В `filePath` указывается полное имя файла, а в `fileObj` — объект `File`, который описывает файл.

В следующем примере создаются два объекта `FileInputStream`, которые задействуют тот же самый файл и каждый из двух конструкторов:

```
FileInputStream f0 = new FileInputStream("/autoexec.bat");
File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

Хотя первый конструктор, вероятно, применяется чаще, второй позволяет внимательно исследовать файл с помощью методов `File`, прежде чем присоединить его к входному потоку. Созданный объект `FileInputStream` также открывается для чтения. В классе `FileInputStream` переопределено несколько методов абстрактного класса `InputStream`. Методы `mark()` и `reset()` не переопределяются, а попытка использования `reset()` на объекте `FileInputStream` приведет к генерации исключения `IOException`.

В приведенном далее примере показано, как читать один байт, массив байтов и поддиапазон массива байтов. Также демонстрируется применение метода `available()` для определения количества оставшихся байтов и метода `skip()` для пропуска нежелательных байтов. В программе организуется чтение собственного файла исходного кода, который должен находиться в текущем каталоге. Обратите внимание, что в нем используется оператор `try` с ресурсами для автоматического закрытия файла, когда он больше не нужен.

```
// Демонстрация использования FileInputStream.
import java.io.*;
class FileInputStreamDemo {
  public static void main(String[] args) {
    int size;
```

```
    // Применить try с ресурсами для закрытия потока данных.
    try ( FileInputStream f =
          new FileInputStream("FileInputStreamDemo.java") ) {
      System.out.println("Всего доступно байтов: " + (size = f.available()));
      int n = size/40;
      System.out.println("Первые " + n +
                         " байтов файла, прочитанные по одному с помощью read()");
      for (int i=0; i < n; i++) {
        System.out.print((char) f.read());
      }
      System.out.println("\nВсе еще доступно байтов: " + f.available());
      System.out.println("Чтение следующих " + n +
                         " байтов с помощью одного вызова read(b[])");
      byte[] b = new byte[n];
      if (f.read(b) != n) {
        System.err.println("Не удалось прочитать " + n + " байтов.");
      }
      System.out.println(new String(b, 0, n));
      System.out.println("\nВсе еще доступно байтов: "
        + (size = f.available()));
      System.out.println("Пропуск половины оставшихся байтов с помощью skip()");
      f.skip(size/2);
      System.out.println("Все еще доступно байтов: " + f.available());
      System.out.println("Чтение " + n/2 + " байтов в конец массива");
      if (f.read(b, n/2, n/2) != n/2) {
        System.err.println("Не удалось прочитать " + n/2 + " байтов.");
      }
      System.out.println(new String(b, 0, b.length));
      System.out.println("\nВсе еще доступно байтов: " + f.available());
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

Вот вывод:

```
Всего доступно байтов: 2125
Первые 53 байтов файла, прочитанные по одному с помощью read()
// Демонстрация использования FileInputStream.

imp
Все еще доступно байтов: 2072
Чтение следующих 53 байтов с помощью одного вызова read(b[])
ort java.io.*;

class FileInputStreamDemo {
  publ
Все еще доступно байтов: 2019
Пропуск половины оставшихся байтов с помощью skip()
Все еще доступно байтов: 1010
Чтение 26 байтов в конец массива
e()) ;

     System.out.
Все еще доступно байтов: 984
```

В этом несколько надуманном примере иллюстрировалось чтение тремя способами, пропуск входных данных и проверка объема данных, доступных в потоке.

На заметку! В предыдущем примере и в других примерах настоящей главы обрабатываются любые исключения ввода-вывода, которые могут возникать, как было описано в главе 13. Ищите в ней детальные сведения и альтернативные варианты.

FileOutputStream

Класс `FileOutputStream` создает объект `OutputStream`, который можно применять для записи байтов в файл. Он реализует интерфейсы `AutoCloseable`, `Closeable` и `Flushable`. Ниже показаны четыре его конструктора:

```
FileOutputStream(String filePath)
FileOutputStream(File fileObj)
FileOutputStream(String filePath, boolean append)
FileOutputStream(File fileObj, boolean append)
```

Конструкторы могут генерировать исключение `FileNotFoundException`. В `filePath` указывается полное имя файла, а в `fileObj` — объект `File`, описывающий файл. Если `append` имеет значение `true`, тогда файл открывается в режиме добавления.

Конструктор класса `FileOutputStream` не полагается на то, что файл уже существует, а создает его перед открытием для вывода при создании объекта. Попытка открытия файла, допускающего только чтение, приведет к генерации исключения.

В следующей программе создается буфер байтов с текстовым образцом. Сначала создается объект `String`, после чего с помощью метода `getBytes()` из него получается эквивалент в виде байтового массива. Затем создаются три файла. Первый файл, `file1.txt`, будет содержать каждый второй байт образца, во втором файле, `file2.txt`, сохранится весь набор байтов, а в третий файл, `file3.txt`, попадет только последняя четверть набора.

```
// Демонстрация использования FileOutputStream.
// В этой программе применяется традиционный подход к закрытию файла.

import java.io.*;

class FileOutputStreamDemo {
  public static void main(String[] args) {
    String source = "Настало время всем порядочным людям\n"
      + " прийти на помощь своей стране\n"
      + " и заплатить надлежащие налоги.";
    byte[] buf = source.getBytes();
    FileOutputStream f0 = null;
    FileOutputStream f1 = null;
    FileOutputStream f2 = null;

    try {
      f0 = new FileOutputStream("file1.txt");
      f1 = new FileOutputStream("file2.txt");
      f2 = new FileOutputStream("file3.txt");
```

```
      // Записать в первый файл.
      for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);
      // Записать во второй файл.
      f1.write(buf);
      // Записать в третий файл.
      f2.write(buf, buf.length-buf.length/4, buf.length/4);
    } catch(IOException e) {
      System.out.println("Возникла ошибка ввода-вывода");
    } finally {
      try {
        if(f0 != null) f0.close();
      } catch(IOException e) {
        System.out.println("Ошибка при закрытии file1.txt");
      }
      try {
        if(f1 != null) f1.close();
      } catch(IOException e) {
        System.out.println("Ошибка при закрытии file2.txt");
      }
      try {
        if(f2 != null) f2.close();
      } catch(IOException e) {
        System.out.println("Ошибка при закрытии file3.txt");
      }
    }
  }
}
```

Давайте посмотрим, что содержат файлы после выполнения программы. Вот содержимое `file1.txt`:

```
Нсаовеяве ояонмлдм рйин ооьсойсрн
изпаиьндеаи аои
```

А вот содержимое `file2.txt`:

```
Настало время всем порядочным людям
 прийти на помощь своей стране
 и заплатить надлежащие налоги.
```

Наконец, вот содержимое `file3.txt`:

```
латить надлежащие налоги.
```

Как упоминалось в комментарии в начале программы, в ней используется традиционный подход к закрытию файла, когда он больше не нужен. Такой подход обязателен во всех версиях Java, предшествующих JDK 7, и широко применяется в унаследованном коде. Легко заметить, что для явного вызова `close()` требуется довольно неуклюжий код, поскольку каждый вызов может генерировать исключение `IOException`, если операция закрытия завершилась неудачно. Программу можно существенно улучшить, используя оператор `try` с ресурсами. Для сравнения ниже показана переделанная версия. Обратите внимание, что она намного короче и рациональнее:

```
// Демонстрация использования FileOutputStream.
// В этой версии программы применяется оператор try с ресурсами.
import java.io.*;
class FileOutputStreamDemo {
  public static void main(String[] args) {
    String source = "Настало время всем порядочным людям\n"
      + " прийти на помощь своей стране\n"
      + " и заплатить надлежащие налоги.";
    byte[] buf = source.getBytes();
    // Использовать для закрытия файлов оператор try с ресурсами.
    try (FileOutputStream f0 = new FileOutputStream("file1.txt");
         FileOutputStream f1 = new FileOutputStream("file2.txt");
         FileOutputStream f2 = new FileOutputStream("file3.txt"))
    {
      // Записать в первый файл.
      for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);
      // Записать во второй файл.
      f1.write(buf);
      // Записать в третий файл.
      f2.write(buf, buf.length-buf.length/4, buf.length/4);
    } catch(IOException e) {
      System.out.println("Возникла ошибка ввода-вывода");
    }
  }
}
```

ByteArrayInputStream

ByteArrayInputStream — реализация потока ввода, использующая в качестве источника байтовый массив. Класс ByteArrayInputStream имеет два конструктора, каждому из которых требуется байтовый массив для предоставления источника данных:

```
ByteArrayInputStream(byte[] array)
ByteArrayInputStream(byte[] array, int start, int numBytes)
```

В array передается источник ввода. Второй конструктор создает объект InputStream из подмножества байтового массива, которое начинается с символа по индексу, указанному в start, и имеет длину numBytes.

Метод close() не оказывает влияния на ByteArrayInputStream, так что нет никакой необходимости вызывать close() на объекте ByteArrayInputStream, но это не будет ошибкой.

В показанном ниже примере создается пара объектов ByteArrayInputStream, которые инициализируются байтовым представлением английского алфавита:

```
// Демонстрация использования ByteArrayInputStream.
import java.io.*;
class ByteArrayInputStreamDemo {
  public static void main(String[] args) {
    String tmp = "abcdefghijklmnopqrstuvwxyz";
    byte[] b = tmp.getBytes();
```

```
    ByteArrayInputStream input1 = new ByteArrayInputStream(b);
    ByteArrayInputStream input2 = new ByteArrayInputStream(b,0,3);
  }
}
```

Объект input1 содержит полный английский алфавит в нижнем регистре, а input2 — только первые три буквы.

В классе ByteArrayInputStream реализованы методы mark() и reset(). Тем не менее, если метод mark() не вызывался, тогда метод reset() будет устанавливать указатель потока на начало потока, которым в данном случае является начало байтового массива, переданного конструктору. В следующем примере иллюстрируется применение метода reset() для двукратного чтения тех же самых входных данных. В этом случае программа читает и печатает буквы "abc" один раз в нижнем регистре и еще один раз в верхнем.

```
import java.io.*;

class ByteArrayInputStreamReset {
  public static void main(String[] args) {
    String tmp = "abc";
    byte[] b = tmp.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(b);

    for (int i=0; i<2; i++) {
      int c;
      while ((c = in.read()) != -1) {
        if (i == 0) {
          System.out.print((char) c);
        } else {
          System.out.print(Character.toUpperCase((char) c));
        }
      }
      System.out.println();
      in.reset();
    }
  }
}
```

В программе сначала производится чтение каждого символа из потока и его вывод в неизменном виде в нижнем регистре. Затем он сбрасывает поток и снова начинает чтение, преобразовывая каждый символ перед выводом в верхний регистр. Вот результат:

```
abc
ABC
```

ByteArrayOutputStream

ByteArrayOutputStream — реализация потока вывода, который использует байтовый массив в качестве места назначения.

Класс ByteArrayOutputStream имеет два конструктора:

```
ByteArrayOutputStream()
ByteArrayOutputStream(int numBytes)
```

Первая форма конструктора создает буфер размером 32 байта. Вторая форма создает буфер с размером, указанным в numBytes. Буфер хранится в защищенном поле buf класса ByteArrayOutputStream. При необходимости размер буфера автоматически увеличивается. Количество байтов, имеющихся в буфере, содержится в защищенном поле count класса ByteArrayOutputStream.

Метод close() не оказывает влияния на ByteArrayOutputStream, так что нет никакой необходимости вызывать close() на объекте ByteArrayOutputStream, но это не будет ошибкой. В показанном ниже примере демонстрируется работа с классом ByteArrayOutputStream:

```
// Демонстрация использования ByteArrayOutputStream.

import java.io.*;

class ByteArrayOutputStreamDemo {
  public static void main(String[] args) {
    ByteArrayOutputStream f = new ByteArrayOutputStream();
    String s = "Эта строка в итоге должна оказаться в массиве";
    byte[] buf = s.getBytes();

    try {
      f.write(buf);
    } catch(IOException e) {
      System.out.println("Ошибка при записи в буфер");
      return;
    }

    System.out.println("Буфер в виде строки:");
    System.out.println(f.toString());
    System.out.println("В массив:");
    byte[] b = f.toByteArray();
    for (int i=0; i<b.length; i++) System.out.print((char) b[i]);

    System.out.println("\nВ поток вывода:");

    // Использовать для управления файловым потоком оператор try с ресурсами.
    try ( FileOutputStream f2 = new FileOutputStream("test.txt") )
    {
      f.writeTo(f2);
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
      return;
    }

    System.out.println("Выполнение переустановки");
    f.reset();

    for (int i=0; i<3; i++) f.write('X');

    System.out.println(f.toString());
  }
}
```

Запуск программы приводит к получению следующего вывода. Обратите внимание, что после вызова reset() три символа X оказываются в начале.

```
Буфер в виде строки:
Эта строка в итоге должна оказаться в массиве
В массив:
Эта строка в итоге должна оказаться в массиве
В поток вывода:
Выполнение переустановки
XXX
```

В этом примере для записи содержимого f в test.txt применяется удобный метод writeTo(). Просмотр содержимого файла test.txt, созданного в предыдущем примере, подтверждает ожидаемый результат:

```
Эта строка в итоге должна оказаться в массиве
```

Фильтрующие байтовые потоки

Фильтрующие потоки данных представляют собой просто оболочки базовых потоков ввода или вывода, которые прозрачно обеспечивают некоторый расширенный уровень функциональности. К таким потокам обычно обращаются методы, ожидающие обобщенный поток, который является суперклассом для фильтрующих потоков. К типичным расширениям относятся буферизация, трансляция символов и преобразование низкоуровневых данных. Фильтрующие байтовые потоки реализованы классами FilterInputStream и FilterOutputStream со следующими конструкторами:

```
FilterOutputStream(OutputStream os)
FilterInputStream(InputStream is)
```

Предоставляемые этими классами методы идентичны методам в InputStream и OutputStream.

Буферизованные байтовые потоки

Для потоков, ориентированных на байты, *буферизованный поток* расширяет класс фильтрующего потока, присоединяя буфер в памяти к потоку ввода-вывода. Такой буфер позволяет выполнять операции ввода-вывода с более чем одним байтом за раз, тем самым повышая производительность. Поскольку буфер доступен, становятся возможными пропуск, маркировка и переустановка потока. Классами буферизованного байтового потока являются BufferedInputStream и BufferedOutputStream. Кроме того, буферизованный поток также реализован классом PushbackInputStream.

BufferedInputStream

Буферизация ввода-вывода — очень распространенный способ оптимизации производительности. Класс BufferedInputStream в Java позволяет помещать любой объект InputStream в оболочку буферизованного потока с целью повышения производительности.

В классе BufferedInputStream определены два конструктора:

```
BufferedInputStream(InputStream inputStream)
BufferedInputStream(InputStream inputStream, int bufSize)
```

Первая форма конструктора создает буферизованный поток со стандартным размером буфера. Во второй форме конструктора размер буфера передается в `bufSize`. Использование размеров, кратных размеру страницы памяти, дисковому блоку и т.д., может оказать существенное позитивное влияние на производительность. Однако имеется зависимость от реализации. Оптимальный размер буфера обычно зависит от операционной системы хоста, объема доступной памяти и конфигурации машины. Чтобы эффективно применять буферизацию, не обязательно нужна такая степень сложности. Хорошей приблизительной оценкой для размера является 8192 байт, и присоединение даже довольно небольшого буфера к потоку ввода-вывода всегда будет удачной идеей. Таким образом, низкоуровневая система может читать блоки данных с диска либо из сети и сохранять результаты в вашем буфере. Таким образом, даже если вы читаете данные побайтно из `InputStream`, то большую часть времени будете манипулировать быстрой памятью.

Буферизация потока ввода также обеспечивает основу, необходимую для поддержки перемещения назад в потоке данных с доступным буфером. Помимо методов `read()` и `skip()`, реализованных в любом `InputStream`, класс `BufferedInputStream` также поддерживает методы `mark()` и `reset()`. Эта поддержка отражена в том, что `BufferedInputStream.markSupported()` возвращает `true`.

В показанном далее примере воспроизводится ситуация, когда можно использовать `mark()`, чтобы запомнить местоположение в потоке ввода, и затем `reset()`, чтобы туда вернуться. В примере выполняется синтаксический анализ потока с целью обнаружения ссылки на HTML-элемент для символа авторского права. Такая ссылка начинается с амперсанда (&) и заканчивается точкой с запятой (;) без промежуточных пробелов. Образец входных данных содержит два амперсанда для демонстрации случая, когда `reset()` происходит, а когда нет.

```
// Использование буферизованного ввода.
import java.io.*;
class BufferedInputStreamDemo {
  public static void main(String[] args) {
    String s = "Конструкция &copy; - символ авторского права, " +
      "но конструкция &copy - нет.\n";
    byte[] buf = s.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    int c;
    boolean marked = false;

    // Использовать для управления файлом оператор try с ресурсами.
    try ( BufferedInputStream f = new BufferedInputStream(in) )
    {
      while ((c = f.read()) != -1) {
        switch(c) {
          case '&':
            if (!marked) {
              f.mark(32);
              marked = true;
```

```
            } else {
              marked = false;
            }
            break;
          case ';':
            if(marked) {
              marked = false;
              System.out.print("(c)");
            } else
              System.out.print((char) c);
            break;
          case ' ':
            if(marked) {
              marked = false;
              f.reset();
              System.out.print("&");
            } else
              System.out.print((char) c);
            break;
          default:
            if(!marked)
              System.out.print((char) c);
            break;
        }
      }
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

Обратите внимание, что в приведенном примере задействован вызов `mark(32)`, который сохраняет метку для следующих 32 прочитанных байтов (чего достаточно для всех ссылок на HTML-элементы). Вот вывод программы:

```
Конструкция (c) - символ авторского права, но конструкция &copy - нет.
```

BufferedOutputStream

Класс `BufferedOutputStream` похож на любой `OutputStream` за исключением того, что метод `flush()` применяется для обеспечения записи буферов данных в буферизуемый поток. Поскольку целью `BufferedOutputStream` является повышение производительности за счет уменьшения количества раз, когда система фактически записывает данные, может понадобиться вызов `flush()`, чтобы немедленно записать любые данные, находящиеся в буфере.

В отличие от буферизованного ввода, буферизация вывода не предоставляет дополнительной функциональности. Буферы для вывода в Java предназначены для повышения производительности. Вот два доступных конструктора класса `BufferedOutputStream`:

```
BufferedOutputStream(OutputStream outputStream)
BufferedOutputStream(OutputStream outputStream, int bufSize)
```

Первая форма конструктора создает буферизованный поток, используя стандартный размер буфера. Во второй форме конструктора в `bufSize` передается размер буфера.

PushbackInputStream

Одним из новых применений буферизации является реализация обратной передачи (pushback). *Обратная передача* позволяет прочитать байт из входного потока и затем вернуть его обратно. Такая идея воплощена в классе `PushbackInputStream`, который обеспечивает механизм быстрого просмотра того, что поступает из входного потока, не нарушая его работу.

В классе `PushbackInputStream` определены следующие конструкторы:

```
PushbackInputStream(InputStream inputStream)
PushbackInputStream(InputStream inputStream, int numBytes)
```

Первая форма конструктора создает объект потока, который позволяет возвращать один байт во входной поток. Вторая форма конструктора создает поток с буфером обратной передачи длиной `numBytes`, что позволяет возвращать во входной поток несколько байтов.

Помимо знакомых методов класса `InputStream` в `PushbackInputStream` предлагается метод `unread()`:

```
void unread(int b)
void unread(byte[] buffer)
void unread(byte buffer, int offset, int numBytes)
```

Первая форма `unread()` обеспечивает обратную передачу младшего байта `b`, который будет байтом, возвращаемым последующим вызовом `read()`. Вторая форма `unread()` выполняет обратную передачу байтов в `buffer`. Третья форма производит обратную передачу `numBytes` байтов, начиная со смещения `offset` в `buffer`. Попытка обратной передачи байта, когда буфер обратной передачи заполнен, приводит к генерации исключения `IOException`.

Ниже приведен пример, в котором показано, каким образом синтаксический анализатор языка программирования может использовать `PushbackInputStream` и `unread()` для проведения различий между операцией сравнения (==) и операцией присваивания (=):

```
// Демонстрация использования unread().

import java.io.*;

class PushbackInputStreamDemo {
  public static void main(String[] args) {
    String s = "if (a == 4) a = 0;\n";
    byte[] buf = s.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    int c;

    try ( PushbackInputStream f = new PushbackInputStream(in) )
    {
      while ((c = f.read()) != -1) {
```

```
        switch(c) {
          case '=':
            if ((c = f.read()) == '=')
              System.out.print(".eq.");
            else {
              System.out.print("<-");
              f.unread(c);
            }
            break;
          default:
            System.out.print((char) c);
            break;
        }
      }
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

Вот вывод, который выдает пример. Обратите внимание, что операция == была заменена конструкцией .eq., а операция = — конструкцией <-.

```
if (a .eq. 4) a <- 0;
```

Внимание! Побочный эффект класса `PushbackInputStream` заключается в том, что он делает недействительными методы `mark()` и `reset()` объекта `InputStream`, который применялся при его создании. Используйте метод `markSupported()` для проверки любого потока, на котором вы собираетесь вызывать `mark()`/`reset()`.

SequenceInputStream

Класс `SequenceInputStream` позволяет объединять множество потоков `InputStream`. Создание `SequenceInputStream` отличается от любого другого `InputStream`. Конструктор `SequenceInputStream` принимает в качестве аргумента либо пару объектов `InputStream`, либо перечисление объектов `InputStream`:

```
SequenceInputStream(InputStream first, InputStream second)
SequenceInputStream(Enumeration <? extends InputStream> streamEnum)
```

В процессе своей работы объект `SequenceInputStream` выполняет запросы на чтение из первого объекта `InputStream`, пока данные в нем не закончатся, а затем переключается на второй объект. В случае перечисления чтение будет продолжаться по всем объектам `InputStream`, пока не будет достигнут конец последнего. По достижении конца каждого файла ассоциированный с ним поток закрывается. Закрытие потока, созданного `SequenceInputStream`, приводит к закрытию всех незакрытых потоков.

Далее приведен простой пример использования класса `SequenceInputStream` для вывода содержимого двух файлов. В демонстрационных целях в программе применяется традиционный прием закрытия файла. В качестве упражнения можете попробовать изменить код, чтобы использовать оператор `try` с ресурсами.

```
// Демонстрация последовательного ввода.
// В этой программе используется традиционный подход к закрытию файла.
import java.io.*;
import java.util.*;

class InputStreamEnumerator implements Enumeration<FileInputStream> {
  private Enumeration<String> files;

  public InputStreamEnumerator(Vector<String> files) {
    this.files = files.elements();
  }

  public boolean hasMoreElements() {
    return files.hasMoreElements();
  }

  public FileInputStream nextElement() {
    try {
      return new FileInputStream(files.nextElement().toString());
    } catch (IOException e) {
      return null;
    }
  }
}

class SequenceInputStreamDemo {
  public static void main(String[] args) {
    int c;
    Vector<String> files = new Vector<String>();

    files.addElement("file1.txt");
    files.addElement("file2.txt");
    files.addElement("file3.txt");
    InputStreamEnumerator ise = new InputStreamEnumerator(files);
    InputStream input = new SequenceInputStream(ise);

    try {
      while ((c = input.read()) != -1)
        System.out.print((char) c);
    } catch(NullPointerException e) {
      System.out.println("Ошибка при открытии файла.");
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    } finally {
      try {
        input.close();
      } catch(IOException e) {
        System.out.println("Ошибка при закрытии SequenceInputStream");
      }
    }
  }
}
```

В этом примере создается объект `Vector`, после чего к нему добавляются три имени файла. Вектор с именами передается конструктору класса `InputStreamEnumerator`, который предназначен для предоставления оболочки вокруг вектора, в которой возвращаемые элементы являются не имена-

ми файлов, а открытыми объектами FileInputStream для файлов с такими именами. Объект SequenceInputStream открывает каждый файл по очереди, и в рассмотренном примере выводится содержимое файлов.

Обратите внимание в методе nextElement(), что если файл не может быть открыт, тогда возвращается null, что приводит к генерации исключения NullPointerException, которое перехватывается в main().

PrintStream

Класс PrintStream предлагает все возможности вывода, которые применялись через файловый дескриптор System.out с самого начала книги. В результате PrintStream является одним из наиболее часто используемых классов Java. Он реализует интерфейсы Appendable, AutoCloseable, Closeable и Flushable.

В классе PrintStream определено несколько конструкторов. Первым делом взглянем на конструкторы, перечисленные ниже:

```
PrintStream(OutputStream outputStream)
PrintStream(OutputStream outputStream, boolean autoFlushingOn)
PrintStream(OutputStream outputStream, boolean autoFlushingOn String charSet)
  throws UnsupportedEncodingException
```

В outputStream указывается открытый объект OutputStream, который будет получать выходные данные. Параметр autoFlushingOn управляет тем, будет ли буфер вывода автоматически сбрасываться каждый раз, когда записывается символ новой строки (\n) или массив байтов, либо когда вызывается метод println(). Если autoFlushingOn имеет значение true, то сбрасывание происходит автоматически, а если false, тогда сбрасывание не будет автоматическим. Первая форма конструктора не обеспечивает автоматическое сбрасывание. Кроме того, можно указать кодировку символов, передав ее имя в charSet.

Следующий набор конструкторов предлагает простой способ создания объекта PrintStream, который записывает свои выходные данные в файл:

```
PrintStream(File outputFile) throws FileNotFoundException
PrintStream(File outputFile, String charSet)
  throws FileNotFoundException, UnsupportedEncodingException
PrintStream(String outputFileName) throws FileNotFoundException
PrintStream(String outputFileName, String charSet)
  throws FileNotFoundException, UnsupportedEncodingException
```

Они позволяют создавать объект PrintStream из объекта File или за счет указания имени файла. В любом случае файл создается автоматически. Ранее существовавший файл с тем же самым именем уничтожается. После создания объект PrintStream направляет все выходные данные в указанный файл. Вдобавок в параметре charSet можно передать имя кодировки символов. Существуют также конструкторы, которые позволяют указывать параметр Charset.

На заметку! Если присутствует диспетчер безопасности, тогда некоторые конструкторы класса `PrintStream` будут генерировать исключение `SecurityException` в случае нарушения безопасности. Имейте в виду, что в версии JDK 17 диспетчер безопасности был объявлен устаревшим и подлежащим удалению.

Класс `PrintStream` поддерживает методы `print()` и `println()` для всех типов, включая `Object`. Если аргумент не относится к примитивному типу, то методы `PrintStream` будут вызывать метод `toString()` объекта и затем отображать результат. Класс `PrintStream` также поддерживает ряд методов `write()` и предоставляет методы для обработки ошибок.

Несколько лет назад в класс `PrintStream` был добавлен очень полезный метод `printf()`. Он позволяет указывать точный формат записываемых данных. Метод `printf()` выполняет форматирование, как описано классом `Formatter`, который обсуждался в главе 21. Затем `printf()` записывает эти данные в вызывающий поток. Хотя форматирование можно выполнить вручную за счет применения `Formatter` напрямую, метод `printf()` облегчает процесс. Кроме того, метод `printf()` аналогичен функции `printf()` языка C/C++, что упрощает преобразование существующего кода C/C++ в Java. Откровенно говоря, метод `printf()` стал долгожданным дополнением Java API, поскольку он значительно упростил вывод форматированных данных на консоль.

Метод `printf()` имеет следующие общие формы:

```
PrintStream printf(String fmtString, Object ... args)
PrintStream printf(Locale loc, String fmtString, Object ... args)
```

Первая форма `printf()` записывает `args` в стандартный поток вывода в формате, указанном в `fmtString`, с использованием стандартной локали. Вторая форма `printf()` позволяет указать локаль. Обе формы возвращают вызывающий объект `PrintStream`.

В целом метод `printf()` работает аналогично методу `format()`, заданному с помощью `Formatter`. Параметр `fmtString` состоит из двух типов элементов. Первый тип образован из символов, которые просто копируются в буфер вывода. Второй тип содержит спецификаторы формата, определяющие способ отображения последующих аргументов, указанных в параметре `args`. Получить полную информацию о форматировании вывода, включая описание спецификаторов формата, можно в главе 20, где обсуждался класс `Formatter`.

Поскольку `System.out` — это `PrintStream`, метод `printf()` допускается вызывать на `System.out`. Таким образом, `printf()` может применять вместо `println()` при записи на консоль всякий раз, когда требуется форматированный вывод. Например, в следующей программе метод `printf()` используется для вывода числовых значений в различных форматах. В прошлом такое форматирование требовало выполнения некоторой работы. С добавлением метода `printf()` задача упростилась.

```
// Демонстрация использования printf().
class PrintfDemo {
  public static void main(String[] args) {
```

```
        System.out.println("Вывод ряда числовых значений в разных форматах.\n");

        System.out.printf("Различные форматы для целых чисел: ");
        System.out.printf("%d %(d %+d %05d\n", 3, -3, 3, 3);

        System.out.println();
        System.out.printf("Стандартный формат для чисел с плавающей точкой: %f\n",
                          1234567.123);
        System.out.printf("Формат для чисел с плавающей точкой, содержащий
запятые: %,f\n", 1234567.123);
        System.out.printf("Стандартный формат для отрицательных чисел с
плавающей точкой: %,f\n", -1234567.123);
        System.out.printf("Вариант формата для отрицательных чисел с
плавающей точкой: %,(f\n", -1234567.123);

        System.out.println();
        System.out.printf("Выстраивание в столбик положительных и
отрицательных чисел:\n");
        System.out.printf("% ,.2f\n% ,.2f\n", 1234567.123, -1234567.123);
    }
}
```

Вот вывод:

```
Вывод ряда числовых значений в разных форматах.

Различные форматы для целых чисел: 3 (3) +3 00003

Стандартный формат для чисел с плавающей точкой: 1234567.123000
Формат для чисел с плавающей точкой, содержащий запятые: 1,234,567.123000
Стандартный формат для отрицательных чисел с плавающей точкой: -1,234,567.123000
Вариант формата для отрицательных чисел с плавающей точкой: (1,234,567.123000)

Выстраивание в столбик положительных и отрицательных чисел:
  1,234,567.12
 -1,234,567.12
```

В классе `PrintStream` также определен метод `format()` со следующими общими формами:

```
PrintStream format(String fmtString, Object ... args)
PrintStream format(Locale loc, String fmtString, Object ... args)
```

Метод `format()` работает в точности как `printf()`.

DataOutputStream и DataInputStream

Классы `DataOutputStream` и `DataInputStream` позволяют записывать данные примитивных типов в поток или читать их из него. Они реализуют соответственно интерфейсы `DataOutput` и `DataInput`, которые определяют методы, преобразующие значения примитивных типов в последовательность байтов либо из нее. Эти потоки упрощают хранение двоичных данных, таких как целые числа или значения с плавающей точкой, в файле. Оба они рассматриваются ниже.

Класс `DataOutputStream` расширяет `FilterOutputStream`, в свою очередь расширяющий `OutputStream`. Помимо реализации `DataOutput` класс `DataOutputStream` также реализует интерфейсы `AutoCloseable`, `Closeable` и `Flushable`.

В DataOutputStream определен следующий конструктор:

```
DataOutputStream(OutputStream outputStream)
```

В outputStream указывается поток вывода, куда будут записываться данные. Когда DataOutputStream закрывается (вызовом close()), базовый поток, заданный в outputStream, тоже автоматически закрывается.

Класс DataOutputStream поддерживает все методы, определенные в его суперклассах. Тем не менее, именно методы, определенные в интерфейсе DataOutput, который он реализует, делают его интересным. В DataOutput определены методы, преобразующие значения примитивных типов в последовательность байтов и затем записывающие их в базовый поток. Ниже приведены примеры таких методов:

```
final void writeDouble(double value) throws IOException
final void writeBoolean(boolean value) throws IOException
final void writeInt(int value) throws IOException
```

Здесь в value указывается значение, записываемое в поток.

Класс DataInputStream является дополнением класса DataOutputStream. Он расширяет FilterInputStream, в свою очередь расширяющий InputStream. Вдобавок к реализации интерфейса DataInput класс DataInputStream также реализует интерфейсы AutoCloseable и Closeable. Он имеет единственный конструктор:

```
DataInputStream(InputStream inputStream)
```

В inputStream указывается поток ввода, из которого будут читаться данные. Когда объект DataInputStream закрывается (вызовом close()), базовый поток, заданный в inputStream, тоже автоматически закрывается.

Подобно DataOutputStream класс DataInputStream поддерживает все методы своих суперклассов, но именно методы, определенные в интерфейсе DataInput, делают его уникальным. Эти методы читают последовательность байтов и преобразуют их в значения примитивных типов. Вот примеры таких методов:

```
final double readDouble() throws IOException
final boolean readBoolean() throws IOException
final int readInt() throws IOException
```

В следующей программе демонстрируется применение классов DataOutputStream и DataInputStream:

```
// Демонстрация использования DataInputStream и DataOutputStream.

import java.io.*;

class DataIODemo {
  public static void main(String[] args) throws IOException {

    // Для начала записать данные.
    try ( DataOutputStream dout =
      new DataOutputStream(new FileOutputStream("Test.dat")) )
    {
      dout.writeDouble(98.6);
      dout.writeInt(1000);
      dout.writeBoolean(true);
```

```
    } catch(FileNotFoundException e) {
      System.out.println("Не удалось открыть выходной файл");
      return;
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
    // Теперь прочитать данные.
    try ( DataInputStream din =
      new DataInputStream(new FileInputStream("Test.dat")) )
    {
      double d = din.readDouble();
      int i = din.readInt();
      boolean b = din.readBoolean();
      System.out.println("Полученные значения: " +
                         d + " " + i + " " + b);
    } catch(FileNotFoundException e) {
      System.out.println("Не удалось открыть входной файл");
      return;
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

Ниже показан вывод:

```
Полученные значения: 98.6 1000 true
```

RandomAccessFile

Класс `RandomAccessFile` инкапсулирует файл с произвольным доступом. Он не является производным от `InputStream` или `OutputStream`. Взамен класс `RandomAccessFile` реализует интерфейсы `DataInput` и `DataOutput`, в которых определены основные методы ввода-вывода. Он также реализует интерфейсы `AutoCloseable` и `Closeable`. Особенность класса `RandomAccessFile` заключается в том, что он поддерживает запросы на позиционирование, т.е. имеется возможность позиционирования файлового указателя внутри файла. Вот два его конструктора:

```
RandomAccessFile(File fileObj, String access) throws  FileNotFoundException
RandomAccessFile(String filename, String access) throws FileNotFoundException
```

В первой форме конструктора в параметре `fileObj` указывается файл, который нужно открыть, в виде объекта `File`. Во второй форме в `filename` передается имя файла. В обоих случаях доступ определяет, какой тип доступа к файлу разрешен. Если это `"r"`, то файл можно читать, но не записывать. Если это `"rw"`, тогда файл открывается в режиме чтения-записи. Если это `"rws"`, то файл открывается для операций чтения-записи, и каждое изменение данных или метаданных файла будет немедленно записано на физическое устройство. Если это `"rwd"`, тогда файл открывается для операций чтения-записи, и каждое изменение данных файла будет немедленно записано на физическое устройство.

Показанный ниже метод seek() используется для установки в файле текущей позиции указателя файла:

```
void seek(long newPos) throws IOException
```

В newPos задается новая позиция в байтах указателя файла, считая от начала файла. После вызова seek() следующая операция чтения или записи выполнится в новой позиции файла.

Класс RandomAccessFile реализует стандартные методы ввода и вывода, которые можно применять для чтения и записи в файлы с произвольным доступом. Он также содержит ряд дополнительных методов, одним из которых является setLength() со следующей сигнатурой:

```
void setLength(long len) throws IOException
```

Метод setLength() устанавливает длину вызываемого файла в соответствии с параметром len. Его можно использовать для удлинения или сокращения файла. Если файл удлиняется, тогда добавленная порция не определена.

Символьные потоки

Хотя классы байтовых потоков обеспечивают достаточную функциональность для обработки операций ввода-вывода любого вида, они не могут напрямую работать с символами Unicode. Поскольку одной из основных целей Java была поддержка философии "написанное однажды выполняется везде, в любое время, всегда", возникла необходимость во включении прямой поддержки ввода-вывода для символов. В этом разделе обсуждаются классы символьного ввода-вывода. Как объяснялось ранее, на верхушке иерархии символьных потоков находятся абстрактные классы Reader и Writer. С них и начнем.

Reader

Reader — абстрактный класс, определяющий модель потокового ввода символов в Java. Он реализует интерфейсы AutoCloseable, Closeable и Readable. Все методы в классе Reader (кроме markSupported()) при возникновении ошибки генерируют исключение IOException.

В табл. 22.4 представлены краткие описания методов Reader.

Таблица 22.4. Методы, определенные в классе Reader

Метод	Описание
abstract void close()	Закрывает источник ввода. Дальнейшие попытки чтения приведут к генерации исключения IOException
void mark (int numChars)	Помещает метку в текущую позицию потока ввода, которая будет оставаться действительной до тех пор, пока не будет прочитано numChars символов

Окончание табл. 22.4

Метод	Описание
`boolean markSupported()`	Возвращает `true`, если методы `mark()`/`reset()` поддерживаются этим потоком
`static Reader nullReader()`	Возвращает открытое, но пустое средство чтения, не содержащее данных. Таким образом, текущая позиция всегда находится в конце средства чтения, и никакие входные данные не могут быть получены. Однако средство чтения можно закрыть
`int read()`	Возвращает целочисленное представление следующего доступного символа в вызывающем потоке ввода. При попытке чтения в конце потока возвращает `-1`
`int read(char[] buffer)`	Пытается прочитать вплоть до `buffer.length` символов в буфер `buffer` и возвращает фактическое количество успешно прочитанных символов. При попытке чтения в конце потока возвращает `-1`
`int read (CharBuffer buffer)`	Пытается прочитать символы в буфер `buffer` и возвращает фактическое количество успешно прочитанных символов. При попытке чтения в конце потока возвращает `-1`
`abstract int read (char[] buffer, int offset, int numChars)`	Пытается прочитать вплоть до `numChars` символов в буфер `buffer`, начиная с `buffer[offset]`, и возвращает фактическое количество успешно прочитанных символов. При попытке чтения в конце потока возвращает `-1`
`boolean ready()`	Возвращает `true`, если следующий запрос на ввод не будет ожидать, или `false` в противном случае
`void reset()`	Переустанавливает указатель ввода на ранее установленную метку
`long skip (long numChars)`	Пропускает `numChars` символов во входных данных и возвращает фактическое количество пропущенных символов
`long transferTo (Writer writer)`	Копирует содержимое вызывающего средства чтения и возвращает количество скопированных символов

Writer

Writer — абстрактный класс, который определяет потоковый вывод символов. Он реализует интерфейсы AutoCloseable, Closeable, Flushable и Appendable. В случае ошибки все методы класса Writer генерируют исключение IOException. В табл. 22.5 представлены краткие описания методов Writer.

Таблица 22.5. Методы, определенные в классе Writer

Метод	Описание
`Writer append(char ch)`	Добавляет ch в конец вызывающего потока вывода. Возвращает ссылку на вызывающий поток
`Writer append (CharSequence chars)`	Добавляет chars в конец вызывающего потока вывода. Возвращает ссылку на вызывающий поток
`Writer append (CharSequence chars, int begin, int end)`	Добавляет поддиапазон chars, начинающийся с begin и заканчивающийся end-1, в конец вызывающего потока вывода. Возвращает ссылку на вызывающий поток
`abstract void close()`	Закрывает поток вывода. Дальнейшие попытки записи приведут к генерации исключения IOException
`abstract void flush()`	Финализирует состояние вывода, так что любые буферы очищаются, т.е. сбрасывает буферы вывода
`static Writer nullWriter()`	Возвращает открытое, но пустое средство записи, куда никакие выходные данные фактически не записывались. Таким образом, методы вывода могут быть вызваны, но они не производят какой-либо вывод. Однако средство записи можно закрыть
`void write(int ch)`	Записывает одиночный символ в вызывающий поток вывода. Обратите внимание, что параметр имеет тип int, позволяя вызывать метод с выражением без необходимости в приведении к типу char. Однако записываются только младшие 16 бит
`void write(char[] buffer)`	Записывает полный символьный массив в вызывающий поток вывода

Окончание табл. 22.5

Метод	Описание
`abstract void write (char[] buffer, int offset, int numChars)`	Записывает поддиапазон длиной `numChars` символов из буфера `buffer` типа массива, начиная с `buffer[offset]`, в вызывающий поток вывода
`void write(String str)`	Записывает `str` в вызывающий поток вывода
`void write(String str, int offset, int numChars)`	Записывает поддиапазон длиной `numChars` символов из строки `str`, начиная с указанного в `offset` смещения

FileReader

Класс `FileReader` создает средство чтения, которое можно применять для чтения содержимого файла. Вот два часто используемых конструктора класса `FileReader`:

```
FileReader(String filePath)
FileReader(File fileObj)
```

Любой из них может сгенерировать исключение `FileNotFoundException`. В `filePath` указывается полное имя файла, а в `fileObj` — объект `File`, описывающий файл.

В следующем примере показано, как читать строки из файла и отображать их на стандартном устройстве вывода. В нем читается собственный файл исходного кода, который должен находиться в текущем каталоге.

```
// Демонстрация использования FileReader.
import java.io.*;
class FileReaderDemo {
  public static void main(String[] args) {
    try ( FileReader fr = new FileReader("FileReaderDemo.java") )
    {
      int c;
      // Прочитать и отобразить содержимое файла.
      while((c = fr.read()) != -1) System.out.print((char) c);
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

FileWriter

Класс `FileWriter` создает объект `Writer`, который можно применять для записи в файл. Ниже показаны четыре часто используемых конструктора:

```
FileWriter(String filePath)
FileWriter(String filePath, boolean append)
FileWriter(File fileObj)
FileWriter(File fileObj, boolean append)
```

Все они способны генерировать исключение `IOException`. В `filePath` передается полное имя файла, а в `fileObj` — объект `File`, описывающий файл. Если `append` имеет значение `true`, тогда выходные данные добавляются в конец файла.

Конструктор класса `FileWriter` не полагается на то, что файл уже существует, а создает его перед открытием для вывода при создании объекта. Попытка открытия файла, допускающего только чтение, приводит к генерации исключения `IOException`.

Следующая программа представляет собой версию примера, рассмотренного ранее при обсуждении класса `FileOutputStream`. В ней формируется буфер с образцом данных, для чего сначала создается объект `String` и затем с помощью метода `getBytes()` из него получается эквивалент в виде символьного массива. Далее создаются три файла. Первый файл, `file1.txt`, будет содержать каждый второй символ образца, во втором файле, `file2.txt`, сохранится весь набор символов, а в третий файл, `file3.txt`, запишется только последняя четверть набора.

```
// Демонстрация использования FileWriter.
import java.io.*;
class FileWriterDemo {
  public static void main(String[] args) throws IOException {
    String source = "Настало время всем порядочным людям\n"
      + " прийти на помощь своей стране\n"
      + " и заплатить надлежащие налоги.";
    char[] buffer = new char[source.length()];
    source.getChars(0, source.length(), buffer, 0);
    try ( FileWriter f0 = new FileWriter("file1.txt");
          FileWriter f1 = new FileWriter("file2.txt");
          FileWriter f2 = new FileWriter("file3.txt") )
    {
      // Записать в первый файл.
      for (int i=0; i < buffer.length; i += 2) {
        f0.write(buffer[i]);
      }
      // Записать во второй файл.
      f1.write(buffer);
      // Записать в третий файл.
      f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
    } catch(IOException e) {
      System.out.println("Возникла ошибка ввода-вывода");
    }
  }
}
```

CharArrayReader

Класс CharArrayReader — это реализация потока ввода, которая применяет в качестве источника символьный массив. В классе CharArrayReader определены два конструктора, каждому из которых требуется символьный массив для предоставления источника данных:

```
CharArrayReader(char[] array)
CharArrayReader(char[] array, int start, int numChars)
```

Здесь array является источником ввода. Второй конструктор создает объект CharArrayReader из подмножества символьного массива, которое начинается с символа по индексу start и имеет длину numChars.

Метод close(), реализованный в CharArrayReader, никаких исключений не генерирует. Причина в том, что он не может потерпеть неудачу.

В следующем примере используется пара объектов CharArrayReader:

```
// Демонстрация использования CharArrayReader.

import java.io.*;

public class CharArrayReaderDemo {
  public static void main(String[] args) {
    String tmp = "abcdefghijklmnopqrstuvwxyz";
    int length = tmp.length();
    char[] c = new char[length];
    tmp.getChars(0, length, c, 0);
    int i;

    try (CharArrayReader input1 = new CharArrayReader(c) )
    {
      System.out.println("Содержимое input1:");
      while((i = input1.read()) != -1) {
        System.out.print((char)i);
      }
      System.out.println();
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }

    try ( CharArrayReader input2 = new CharArrayReader(c, 0, 5) )
    {
      System.out.println("Содержимое input2:");
      while((i = input2.read()) != -1) {
        System.out.print((char)i);
      }
      System.out.println();
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

Объект input1 создан с применением полного английского алфавита в нижнем регистре, а input2 содержит только первые пять букв. Вот результат:

```
Содержимое input1:
abcdefghijklmnopqrstuvwxyz
Содержимое input2:
abcde
```

CharArrayWriter

Класс CharArrayWriter — это реализация потока вывода, который использует массив в качестве места назначения. В классе CharArrayWriter определены два конструктора:

```
CharArrayWriter()
CharArrayWriter(int numChars)
```

В первой форме конструктора создается буфер со стандартным размером, а во второй форме — с размером, который указан в numChars. Буфер хранится в поле buf класса CharArrayWriter. При необходимости размер буфера будет автоматически увеличиваться. Количество символов в буфере содержится в поле count класса CharArrayWriter. И buf, и count являются защищенными полями.

Метод close() не оказывает влияния на CharArrayWriter.

В показанном далее примере демонстрируется применение класса CharArrayWriter за счет переработки программы, приведенной ранее для ByteArrayOutputStream. Программа производит тот же результат, что и предыдущая версия.

```
// Демонстрация использования CharArrayWriter.
import java.io.*;
class CharArrayWriterDemo {
  public static void main(String[] args) throws IOException {
    CharArrayWriter f = new CharArrayWriter();
    String s = "Эта строка в итоге должна оказаться в массиве";
    char[] buf = new char[s.length()];
    s.getChars(0, s.length(), buf, 0);
    try {
      f.write(buf);
    } catch(IOException e) {
      System.out.println("Ошибка при записи в буфер");
      return;
    }
    System.out.println("Буфер в виде строки:");
    System.out.println(f.toString());
    System.out.println("В массив:");
    char[] c = f.toCharArray();
    for (int i=0; i<c.length; i++) {
      System.out.print(c[i]);
    }
    System.out.println("\nВ объект FileWriter:");
```

```
    // Использовать для управления файловым потоком оператор try с ресурсами.
    try ( FileWriter f2 = new FileWriter("test.txt") )
    {
      f.writeTo(f2);
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
    System.out.println("Выполнение переустановки");
    f.reset();
    for (int i=0; i<3; i++) f.write('X');
    System.out.println(f.toString());
  }
}
```

BufferedReader

Класс BufferedReader повышает производительность за счет буферизации ввода. Он имеет два конструктора:

```
BufferedReader(Reader inputStream)
BufferedReader(Reader inputStream, int bufSize)
```

Первая форма конструктора создает буферизованный символьный поток с использованием стандартного размера буфера. Во второй форме размер буфера передается в bufSize.

Закрытие объекта BufferedReader также приводит к закрытию базового потока, указанного в inputStream.

Как и в случае с потоком, ориентированным на байты, буферизация символьного потока ввода обеспечивает основу, необходимую для поддержки перемещения назад в потоке в пределах доступного буфера. Для этого в BufferedReader реализованы методы mark() и reset(), а вызов BufferedReader.markSupported() возвращает true. Относительно недавним добавлением в BufferedReader стал метод lines(), который возвращает ссылку типа Stream на последовательность строк, прочитанных средством чтения. (Класс Stream является частью потокового API, обсуждаемого в главе 30.)

Ниже показан переработанный пример применения BufferedInputStream, который рассматривался ранее. Теперь в нем используется символьный поток BufferedReader, а не буферизованный байтовый поток. Как и прежде, с применением методов mark() и reset() выполняется синтаксический анализ потока с целью обнаружения ссылки на HTML-элемент для символа авторского права. Такая ссылка начинается с амперсанда (&) и заканчивается точкой с запятой (;) без промежуточных пробелов. Образец входных данных содержит два амперсанда для демонстрации случая, когда reset() происходит, а когда нет. Вывод будет таким же, как в предыдущей версии.

```
// Использование буферизованного ввода.
import java.io.*;
class BufferedReaderDemo {
```

```
public static void main(String[] args) throws IOException {
  String s = "Конструкция &copy; - символ авторского права, " +
    "но конструкция &copy - нет.\n";
  char[] buf = new char[s.length()];
  s.getChars(0, s.length(), buf, 0);
  CharArrayReader in = new CharArrayReader(buf);
  int c;
  boolean marked = false;
  try ( BufferedReader f = new BufferedReader(in) )
  {
    while ((c = f.read()) != -1) {
      switch(c) {
        case '&':
          if (!marked) {
            f.mark(32);
            marked = true;
          } else {
            marked = false;
          }
          break;
        case ';':
          if(marked) {
            marked = false;
            System.out.print("(c)");
          } else
            System.out.print((char) c);
          break;
        case ' ':
          if(marked) {
            marked = false;
            f.reset();
            System.out.print("&");
          } else
            System.out.print((char) c);
          break;
        default:
          if(!marked)
            System.out.print((char) c);
          break;
      }
    }
    catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

BufferedWriter

Класс BufferedWriter представляет собой Writer с буферизацией вывода. За счет использования BufferedWriter можно повысить производительность, уменьшая фактическое количество операций физической записи данных на устройство вывода.

В классе BufferedWriter определены два конструктора:

```
BufferedWriter(Writer outputStream)
BufferedWriter(Writer outputStream, int bufSize)
```

Первая форма конструктора создает буферизованный поток с применением буфера стандартного размера. Для второй формы размер буфера передается в bufSize.

PushbackReader

Класс PushbackReader позволяет возвращать во входной поток один или несколько символов, что позволяет заглядывать вперед во входном потоке. Вот два его конструктора:

```
PushbackReader(Reader inputStream)
PushbackReader(Reader inputStream, int bufSize)
```

Первая форма конструктора создает буферизованный поток, который позволяет возвращать обратно один символ. Во второй форме размер буфера обратной передачи указывается в bufSize.

Закрытие объекта PushbackReader закрывает также базовый поток в inputStream. Класс PushbackReader предоставляет метод unread(), который возвращает один или несколько символов в вызывающий входной поток. Он имеет три формы:

```
void unread(int ch) throws IOException
void unread(char[] buffer) throws IOException
void unread(char[] buffer, int offset, int numChars) throws IOException
```

Первая форма unread() обеспечивает обратную передачу символа, указанного в ch, который будет символом, возвращаемым последующим вызовом read(). Вторая форма возвращает символы в buffer. Третья форма производит обратную передачу numChars символов, начиная со смещения offset в buffer. Попытка обратной передачи символа, когда буфер обратной передачи заполнен, приводит к генерации исключения IOException.

Ниже приведена переработанная версия предыдущего примера с классом PushbackInputStream, в которой класс PushbackInputStream заменен PushbackReader. Как и прежде, здесь показано, каким образом синтаксический анализатор языка программирования может использовать поток с обратной передачей для проведения различий между операцией сравнения (==) и операцией присваивания (=).

```
// Демонстрация использования unread().

import java.io.*;

class PushbackReaderDemo {
  public static void main(String[] args) {
    String s = "if (a == 4) a = 0;\n";
    char[] buf = new char[s.length()];
    s.getChars(0, s.length(), buf, 0);
    CharArrayReader in = new CharArrayReader(buf);
```

```
    int c;
    try ( PushbackReader f = new PushbackReader(in) )
    {
      while ((c = f.read()) != -1) {
        switch(c) {
          case '=':
            if ((c = f.read()) == '=')
              System.out.print(".eq.");
            else {
              System.out.print("<-");
              f.unread(c);
            }
            break;
          default:
            System.out.print((char) c);
            break;
        }
      }
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

PrintWriter

Класс `PrintWriter` является по существу версией `PrintStream`, ориентированной на символы. Он реализует интерфейсы `Appendable`, `AutoCloseable`, `Closeable` и `Flushable`. В классе `PrintWriter` определено несколько конструкторов. Для начала рассмотрим следующие конструкторы:

```
PrintWriter(OutputStream outputStream)
PrintWriter(OutputStream outputStream, boolean autoFlushingOn)
PrintWriter(Writer outputStream)
PrintWriter(Writer outputStream, boolean autoFlushingOn)
```

В `outputStream` указывается открытый объект `OutputStream`, который будет получать выходные данные. Параметр `autoFlushingOn` управляет тем, будет ли буфер вывода автоматически сбрасываться при каждом вызове `println()`, `printf()` или `format()`. Если `autoFlushingOn` имеет значение `true`, то сбрасывание происходит автоматически, а если `false`, тогда сбрасывание не будет автоматическим. Конструкторы, в которых нет параметра `autoFlushingOn`, не обеспечивают автоматическое сбрасывание.

Следующий набор конструкторов предлагает простой способ создания объекта `PrintWriter`, который записывает свои выходные данные в файл.

```
PrintWriter(File outputFile) throws FileNotFoundException
PrintWriter(File outputFile, String charSet)
  throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(String outputFileName) throws FileNotFoundException
PrintWriter(String outputFileName, String charSet)
  throws FileNotFoundException, UnsupportedEncodingException
```

Они позволяют создавать объект PrintWriter из объекта File или за счет указания имени файла. В любом случае файл создается автоматически, а ранее существовавший файл с тем же самым именем уничтожается. После создания объект PrintWriter направляет все выходные данные в указанный файл. Имеется возможность задать кодировку символов, передав ее имя в charSet. Есть также конструкторы, которые позволяют указать параметр типа Charset.

Класс PrintWriter поддерживает методы print() и println() для всех типов, включая Object. Если аргумент не относится к примитивному типу, тогда методы класса PrintWriter будут вызывать метод toString() объекта и выводить результат.

Кроме того, класс PrintWriter поддерживает метод printf(). Он работает аналогично методу printf() в описанном ранее классе PrintStream: позволяет указывать точный формат данных. Вот как выглядят объявления printf() в PrintWriter:

```
PrintWriter printf(String fmtString, Object ... args)
PrintWriter printf(Locale loc, String fmtString, Object ... args)
```

Первая форма printf() записывает args в стандартный поток вывода в формате, заданном посредством fmtString, с применением стандартной локали. Вторая форма позволяет указать локаль. Обе формы printf() возвращают вызывающий объект PrintWriter. Вдобавок поддерживается и метод format(), который имеет следующие общие формы:

```
PrintWriter format(String fmtString, Object ... args)
PrintWriter format(Locale loc, String fmtString, Object ... args)
```

Он работает в точности как printf().

Класс Console

Класс Console используется для чтения и записи на консоль, если она существует. Он реализует интерфейс Flushable. Класс Console удобен в первую очередь тем, что большая часть его функциональности доступна через System.in и System.out. Однако он может упростить некоторые типы взаимодействия с консолью, особенно при чтении строк с консоли.

Конструкторы в классе Console отсутствуют. Взамен объект Console получается вызовом System.console():

```
static Console console()
```

Если консоль доступна, тогда возвращается ссылка на нее, а иначе null. Консоль не будет доступной во всех случаях. Таким образом, при возвращении null консольный ввод-вывод недоступен. В классе Console определены методы, кратко описанные в табл. 22.6. Обратите внимание, что методы ввода, такие как readLine(), генерируют исключение IOError, если возникает ошибка ввода. Класс IOError является подклассом Error и указывает на сбой ввода-вывода, который программой не контролируется, а потому исключение IOError обычно перехватываться не будет. Откровенно говоря, если при доступе к консоли возникает исключение IOError, то обычно это означает, что произошел катастрофический системный отказ.

Таблица 22.6. Методы, определенные в классе `Console`

Метод	Описание
`Charset charset()`	Получает объект `Charset`, ассоциированный с консолью, и возвращает результат. (Добавлен в версии JDK 17.)
`void flush()`	Инициирует физическую запись на консоль буферизованных выходных данных
`Console format (String fmtString, Object ... args)`	Записывает `args` на консоль с использованием формата, указанного в `fmtString`
`Console printf (String fmtString, Object ... args)`	Записывает `args` на консоль с использованием формата, указанного в `fmtString`
`Reader reader()`	Возвращает ссылку на объект `Reader`, подключенный к консоли
`String readLine()`	Читает и возвращает строку, введенную с клавиатуры. Ввод останавливается, когда пользователь нажимает клавишу <Enter>. Если достигнут конец консольного потока ввода, тогда возвращается `null`. В случае сбоя генерируется исключение `IOError`
`String readLine (String fmtString, Object ... args)`	Отображает строку приглашения на ввод, сформатированную так, как указано в `fmtString` и `args`, после чего читает и возвращает строку, введенную с клавиатуры. Ввод останавливается, когда пользователь нажимает клавишу <Enter>. Если достигнут конец консольного потока ввода, тогда возвращается `null`. В случае сбоя генерируется исключение `IOError`
`char[] readPassword()`	Читает строку, введенную с клавиатуры. Ввод останавливается, когда пользователь нажимает клавишу <Enter>. Строка не отображается. Если достигнут конец консольного потока ввода, тогда возвращается `null`. В случае сбоя генерируется исключение `IOError`
`char[] readPassword (String fmtString, Object ... args)`	Отображает строку приглашения на ввод, сформатированную так, как указано в `fmtString` и `args`, после чего читает и возвращает строку, введенную с клавиатуры. Ввод останавливается, когда пользователь нажимает клавишу <Enter>. Строка не отображается. Если достигнут конец консольного потока ввода, тогда возвращается `null`. В случае сбоя генерируется исключение `IOError`
`PrintWriter writer()`	Возвращает ссылку на объект `Writer`, подключенный к консоли

Обратите особое внимание на методы `readPassword()`. Они позволяют приложению читать пароль, не повторяя на экране вводимый текст. После чтения паролей понадобится "обнулить" как массив, содержащий введенную пользователем строку, так и массив, содержащий пароль, с которым сравнивается строка. Такой прием снижает вероятность того, что вредоносная программа сможет получить пароль путем сканирования памяти.

Ниже в примере демонстрируется применение класса `Console`:

```
// Демонстрация использования Console.

import java.io.*;

class ConsoleDemo {
  public static void main(String[] args) {
    String str;
    Console con;

    // Получить ссылку на консоль.
    con = System.console();
    // Если консоль недоступна, тогда закончить работу.
    if(con == null) return;

    // Прочитать строку и затем отобразить ее.
    str = con.readLine("Введите строку: ");
    con.printf("Введенная строка: %s\n", str);
  }
}
```

Вот пример вывода, полученного из программы:

```
Введите строку: Строка для тестирования.
Введенная строка: Строка для тестирования.
```

Сериализация

Сериализация — это процесс записи состояния объекта в байтовый поток. Она полезна, когда нужно сохранить состояние программы в постоянном хранилище, скажем, в файле. Позже сериализированные объекты можно восстановить с помощью процесса десериализации.

Сериализация также необходима для реализации *удаленного вызова методов* (Remote Method Invocation — RMI). Механизм RMI позволяет объекту Java на одной машине вызывать метод объекта Java на другой машине. Удаленному методу в качестве аргумента может быть предоставлен объект. Отправляющая машина сериализирует объект и передает его. Принимающая машина десериализует объект. (Дополнительная информация об RMI содержится в главе 31.)

Предположим, что сериализуемый объект имеет ссылки на другие объекты, которые в свою очередь имеют ссылки на дополнительные объекты. Такой набор объектов и отношения между ними образуют ориентированный граф. В этом графе объектов также могут присутствовать циклические ссылки, т.е. объект X может содержать ссылку на объект Y, а объект Y — обратную ссылку на объект X. Кроме того, объекты могут иметь ссылки на самих

себя. Средства сериализации и десериализации объектов были разработаны для правильной работы в сценариях подобного рода. Попытка сериализации объекта в верхней части графа объектов приведет к тому, что все остальные объекты, на которые есть ссылки, будут рекурсивно обнаружены и сериализированы. Точно так же в процессе десериализации все эти объекты и их ссылки корректно восстанавливаются. Важно отметить, что сериализация и десериализация могут повлиять на безопасность, особенно в том, что касается десериализации элементов, которые не вызывают доверия. Поскольку тема безопасности выходит за рамки настоящей книги, актуальную информацию по данной теме ищите в документации по Java. Далее следует обзор интерфейсов и классов, поддерживающих сериализацию.

Serializable

Средства сериализации позволяют сохранять и восстанавливать только объекты, реализующие интерфейс `Serializable`. В интерфейсе `Serializable` никакие члены не определены. Он просто служит для указания на то, что класс можно сериализировать. Если класс сериализуемый, то все его подклассы тоже сериализуемые.

Как правило, при сериализации сохраняются все переменные экземпляра. Тем не менее, средства сериализации не сохраняют переменные, объявленные как `transient`. Кроме того, не сохраняются и статические переменные. (Также есть возможность явно указывать, какие переменные будут сохранены, с использованием массива `serialPersistentFields`.)

Externalizable

Средства сериализации и десериализации Java спроектированы таким образом, что большая часть работы по сохранению и восстановлению состояния объекта выполняется автоматически. Однако бывают случаи, когда программисту может требоваться контроль над этими процессами. Например, иногда желательно применять методики сжатия или шифрования. Для таких ситуаций предназначен интерфейс `Externalizable`.

В интерфейсе `Externalizable` определены два метода:

```
void readExternal(ObjectInput inStream)
  throws IOException, ClassNotFoundException

void writeExternal(ObjectOutput outStream)
  throws IOException
```

Здесь `inStream` — байтовый поток, из которого должен быть прочитан объект, а `outStream` — байтовый поток, куда объект должен быть записан.

ObjectOutput

Интерфейс `ObjectOutput` расширяет интерфейсы `DataOutput` и `AutoCloseable` и поддерживает сериализацию объектов. В нем определены методы, кратко описанные в табл. 22.7.

Таблица 22.7. Методы, определенные в интерфейсе `ObjectOutput`

Метод	Описание
`void close()`	Закрывает вызывающий поток. Дальнейшие попытки записи приведут к генерации исключения `IOException`
`void flush()`	Финализирует состояние вывода, так что любые буферы очищаются, т.е. сбрасывает буферы вывода
`void write(byte[] buffer)`	Записывает в вызывающий поток байтовый массив
`void write(byte[] buffer, int offset, int numBytes)`	Записывает в вызывающий поток поддиапазон длиной `numBytes` байтов из буфера `buffer` типа массива, начиная с `buffer[offset]`
`void write(int b)`	Записывает в вызывающий поток одиночный байт, который является младшим байтом `b`
`void writeObject(Object obj)`	Записывает в вызывающий поток объект `obj`

Обратите особое внимание на метод `writeObject()`. Он вызывается для сериализации объекта. Все методы в случае возникновения ошибки генерируют исключение `IOException`.

ObjectOutputStream

Класс `ObjectOutputStream` расширяет класс `OutputStream` и реализует интерфейс `ObjectOutput`. Он отвечает за запись объектов в поток. Ниже показан один конструктор класса `ObjectOutputStream`:

```
ObjectOutputStream(OutputStream outStream) throws IOException
```

Аргумент `outStream` представляет поток вывода, в который будут записываться сериализированные объекты. Закрытие `ObjectOutputStream` приводит к автоматическому закрытию лежащего в основе потока, указанного в `outStream`.

В табл. 22.8 кратко описаны избранные методы класса `ObjectOutputStream`. При возникновении ошибки они будут генерировать исключение `IOException`. В `ObjectOuputStream` также есть вложенный класс по имени `PutField`, который упрощает запись постоянных полей, но его обсуждение выходит за рамки настоящей книги.

Таблица 22.8. Избранные методы, определенные в классе `ObjectOutputStream`

Метод	Описание
`void close()`	Закрывает вызывающий поток. Дальнейшие попытки записи приведут к генерации исключения `IOException`. Лежащий в основе поток тоже закрывается
`void flush()`	Финализирует состояние вывода, так что любые буферы очищаются, т.е. сбрасывает буферы вывода
`void write(byte[] buffer)`	Записывает в вызывающий поток байтовый массив
`void write(byte[] buffer, int offset, int numBytes)`	Записывает в вызывающий поток поддиапазон длиной `numBytes` байтов из буфера `buffer` типа массива, начиная с `buffer[offset]`
`void write(int b)`	Записывает в вызывающий поток одиночный байт, который является младшим байтом `b`
`void writeBoolean (boolean b)`	Записывает в вызывающий поток значение типа `boolean`
`void writeByte(int b)`	Записывает в вызывающий поток значение типа `byte`, которое является младшим байтом `b`
`void writeBytes (String str)`	Записывает в вызывающий поток байты, представляющие строку `str`
`void writeChar(int c)`	Записывает в вызывающий поток символ
`void writeChars (String str)`	Записывает в вызывающий поток символы в строке `str`
`void writeDouble (double d)`	Записывает в вызывающий поток значение типа `double`
`void writeFloat(float f)`	Записывает в вызывающий поток значение типа `float`
`void writeInt(int i)`	Записывает в вызывающий поток значение типа `int`
`void writeLong(long l)`	Записывает в вызывающий поток значение типа `long`
`final void writeObject(Object obj)`	Записывает в вызывающий поток объект `obj`
`void writeShort(int i)`	Записывает в вызывающий поток значение типа `short`

ObjectInput

Интерфейс ObjectInput расширяет интерфейсы DataInput и AutoCloseable и определяет методы, кратко описанные в табл. 22.9. Он поддерживает сериализацию объектов.

Обратите особое внимание на метод readObject(), который вызывается для десериализации объекта. В случае возникновения ошибки все методы генерируют исключение IOException. Метод readObject() также может генерировать исключение ClassNotFoundException.

Таблица 22.9. Методы, определенные в интерфейсе ObjectInput

Метод	Описание
int available()	Возвращает количество байтов, доступных в текущий момент внутри буфера ввода
void close()	Закрывает вызывающий поток. Дальнейшие попытки чтения приведут к генерации исключения IOException
int read()	Возвращает целочисленное представление следующего байта, доступного во входных данных. При попытке чтения в конце потока возвращает -1
int read(byte[] buffer)	Пытается прочитать вплоть до buffer.length байтов из буфера buffer и возвращает количество успешно прочитанных байтов. При попытке чтения в конце потока возвращает -1
int read(byte[] buffer, int offset, int numBytes)	Пытается прочитать вплоть до numBytes из буфера buffer, начиная с buffer[offset], и возвращает количество успешно прочитанных байтов. При попытке чтения в конце потока возвращает -1
Object readObject()	Читает объект из вызывающего потока
long skip(long numBytes)	Игнорирует (т.е. пропускает) в вызывающем потоке numBytes байтов и возвращает фактическое количество пропущенных байтов

ObjectInputStream

Класс ObjectInputStream расширяет класс InputStream и реализует интерфейс ObjectInput. Он отвечает за чтение объектов из потока. Далее приведен один конструктор класса ObjectInputStream:

```
ObjectInputStream(InputStream inStream) throws IOException
```

Аргумент inStream представляет поток ввода, из которого должны читаться сериализированные объекты. Закрытие объекта ObjectInputStream приводит к автоматическому закрытию лежащего в основе потока, указанного в inStream.

В табл. 22.10 кратко описаны избранные методы класса ObjectInputStream. При возникновении ошибки они будут генерировать исключение IOException. В ObjectInputStream также есть вложенный класс по имени GetField, который упрощает чтение постоянных полей, но его обсуждение выходит за рамки настоящей книги.

Таблица 22.10. Часто используемые методы, определенные в классе ObjectInputStream

Метод	Описание
`int available()`	Возвращает количество байтов, доступных в текущий момент внутри буфера ввода
`void close()`	Закрывает вызывающий поток. Дальнейшие попытки чтения приведут к генерации исключения IOException. Лежащий в основе поток тоже закрывается
`int read()`	Возвращает целочисленное представление следующего байта, доступного во входных данных. При попытке чтения в конце потока возвращает -1
`int read(byte[] buffer, int offset, int numBytes)`	Пытается прочитать вплоть до numBytes из буфера buffer, начиная с buffer[offset], и возвращает количество успешно прочитанных байтов. При попытке чтения в конце потока возвращает -1
`Boolean readBoolean()`	Читает из вызывающего потока и возвращает значение типа boolean
`byte readByte()`	Читает из вызывающего потока и возвращает значение типа byte
`char readChar()`	Читает из вызывающего потока и возвращает значение типа char
`double readDouble()`	Читает из вызывающего потока и возвращает значение типа double
`float readFloat()`	Читает из вызывающего потока и возвращает значение типа float
`void readFully(byte[] buffer)`	Читает из буфера buffer.length байтов. Возвращает управление только после того, как прочитаны все байты
`void readFully(byte[] buffer, int offset, int numBytes)`	Читает из буфера numBytes байтов, начиная с buffer[offset]. Возвращает управление только после того, как прочитано numBytes байтов

Окончание табл. 22.10

Метод	Описание
`int readInt()`	Читает из вызывающего потока и возвращает значение типа `int`
`long readLong()`	Читает из вызывающего потока и возвращает значение типа `long`
`final Object readObject()`	Читает из вызывающего потока и возвращает объект
`short readShort()`	Читает из вызывающего потока и возвращает значение типа `short`
`int readUnsignedByte()`	Читает из вызывающего потока и возвращает значение типа `byte` без знака
`int readUnsignedShort()`	Читает из вызывающего потока и возвращает значение типа `short` без знака

Начиная с версии JDK 9, класс `ObjectInputStream` содержит методы `getObjectInputFilter()` и `setObjectInputFilter()`, которые поддерживают фильтрацию потоков ввода объектов посредством добавленных в JDK 9 типов `ObjectInputFilter`, `ObjectInputFilter.FilterInfo`, `ObjectInputFilter.Config` и `ObjectInputFilter.Status`. Фильтрация обеспечивает определенный контроль над десериализацией.

Пример сериализации

В следующей программе иллюстрируется базовый механизм сериализации и десериализации объектов. Сначала создается объект класса `MyClass`, имеющий три переменных экземпляра с типами `String`, `int` и `double`. Они представляют информацию, которую необходимо сохранить и восстановить.

Далее создается объект `FileOutputStream`, который ссылается на файл по имени `"serial"`, и для этого файлового потока создается `ObjectOutputStream`. Затем с применением метода `writeObject()` класса `ObjectOutputStream` объект сериализируется. Поток вывода объектов сбрасывается и закрывается.

Наконец, создается объект `FileInputStream`, который ссылается на файл по имени `"serial"`, и для данного файлового потока создается `ObjectInputStream`. С использованием метода `readObject()` класса `ObjectInputStream` объект десериализируется. В заключение поток ввода объекта закрывается. Обратите внимание, что класс `MyClass` реализует интерфейс `Serializable`. Если не реализовать `Serializable`, тогда сгенерируется исключение `NotSerializableException`. Попробуйте поэкспериментировать с приведенной программой, объявив некоторые переменные экземпляра `MyClass` как `transient`. Такие данные во время сериализации сохраняться не будут.

```
// Демонстрация использования сериализации.
import java.io.*;
public class SerializationDemo {
  public static void main(String[] args) {

    // Сериализация объекта.
    try ( ObjectOutputStream objOStrm =
         new ObjectOutputStream(new FileOutputStream("serial")) )
    {
      MyClass object1 = new MyClass("Hello", -7, 2.7e10);
      System.out.println("object1: " + object1);
      objOStrm.writeObject(object1);
    }
    catch(IOException e) {
      System.out.println("Во время сериализации возникло исключение: " + e);
    }
    // Десериализация объекта.
    try ( ObjectInputStream objIStrm =
         new ObjectInputStream(new FileInputStream("serial")) )
    {
      MyClass object2 = (MyClass)objIStrm.readObject();
      System.out.println("object2: " + object2);
    }
    catch(Exception e) {
      System.out.println("Во время десериализации возникло исключение: " + e);
    }
  }
}

class MyClass implements Serializable {
  String s;
  int i;
  double d;

  public MyClass(String s, int i, double d) {
    this.s = s;
    this.i = i;
    this.d = d;
  }

  public String toString() {
    return "s=" + s + "; i=" + i + "; d=" + d;
  }
}
```

Вывод программы подтверждает, что переменные экземпляра `object1` и `object2` идентичны:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

Обычно желательно, чтобы классы, предназначенные для сериализации, определяли константу `serialVersionUID` как закрытый член `static`, `final` и `long`. Хотя в Java это значение определяется автоматически (как в случае с `MyClass` в предыдущем примере), в реальных приложениях гораздо лучше определять его явно.

В предыдущем примере был продемонстрирован базовый механизм, применяемый для записи и чтения сериализированных данных. Еще одним ключевым средством, связанным с сериализацией, является *фильтр десериализации*, который предоставляет определенную степень контроля над процессом десериализации. Хотя фильтры десериализации могут быть довольно сложными, добавить к объекту `ObjectInputStream` простой фильтр очень легко. В показанном ниже примере иллюстрируются общие шаги.

В программе `SerializationDemo` такая последовательность добавляет фильтр десериализации в код, который читает объект `MyClass`. Она гарантирует, что `objIStrm` будет десериализировать только объект `MyClass`. Попытка десериализации объекта любого другого класса приведет к генерации исключения во время выполнения.

```
// Десериализация объекта с использованием фильтра.
try ( ObjectInputStream objIStrm =
     new ObjectInputStream(new FileInputStream("serial")) )
{
  // Создать и добавить простой фильтр десериализации.
  ObjectInputFilter myfilter =
    ObjectInputFilter.Config.createFilter("MyClass;!*");
  objIStrm.setObjectInputFilter(myfilter);
  MyClass object2 = (MyClass)objIStrm.readObject();
  System.out.println("object2: " + object2);
}
catch(Exception e) {
  System.out.println("Во время десериализации возникло исключение: " + e);
}
```

Вот основные строки:

```
ObjectInputFilter myfilter =
  ObjectInputFilter.Config.createFilter("MyClass;!*");
objIStrm.setObjectInputFilter(myfilter);
```

Объект `ObjectInputFilter` создается путем вызова статического метода `createFilter()`, определенного во вложенном классе `ObjectInputFilter.Config`. Метод `createFilter()` позволяет указать строковый шаблон, помогающий проверить достоверность ввода. Например, можно указать одно или несколько имен классов, для которых будет разрешена сериализация, разделяя имена точкой с запятой. В рассматриваемом случае задается `MyClass`. Шаблон `!*` указывает на то, что все остальные классы должны быть отклонены. В результате десериализация разрешена только для экземпляров `MyClass`. В целом помещение `!` перед именем класса приводит к тому, что класс отклоняется. Звездочка (`*`) является групповым символом, который соответствует всем именам классов. Созданный фильтр ассоциируется с `objIStrm` вызовом метода `setObjectInputFilter()`, после чего фильтр потока будет активным для потока ввода.

В предыдущем примере был определен фильтр, специфичный для `objIStrm`. Таким образом, другой объект `ObjectInputStream` не будет иметь такой же фильтр. Тем не менее, можно определить фильтр, который будет использо-

ваться всеми объектами ObjectInputStream. Он называется фильтром *на уровне машины JVM* и устанавливается вызовом метода setSerialFilter() в ObjectInputFilter.Config. Например, после следующей строки кода указанный фильтр будут применять все объекты ObjectInputStream:

```
ObjectInputFilter.Config.setSerialFilter(myfilter);
```

Важно понимать, что независимо от того, какой фильтр используется — на уровне машины JVM или специфический для потока, — он должен быть установлен перед чтением из потока. Кроме того, когда применяется фильтр на уровне JVM, его необходимо установить до создания объекта ObjectInputStream. Важно не забывать, что фильтр можно устанавливать только один раз.

При использовании фильтра можно также проверять разнообразные ограничения ресурсов. Ограничения указываются с помощью шаблонов maxdepth, maxrefs, maxbytes и maxarray, которые обязаны включать знак = и за ним значение. Например, ограничения maxbytes определяют максимальную длину потока ввода. Ниже предыдущий фильтр переделан так, чтобы максимальная длина потока ввода составляла 80 байтов:

```
ObjectInputFilter myfilter =
   ObjectInputFilter.Config.createFilter("MyClass;!*;maxbytes=80");
```

Теперь любой объект ObjectInputStream, предоставляющий более 80 байтов, будет отклонен.

В дополнение к только что рассмотренным примерам доступно несколько других вариантов шаблона фильтра. Дополнительные сведения о фильтрах десериализации ищите в документации по Java. И последнее замечание: доступны комментарии к документации, относящиеся к сериализации. (Комментарии к документации обсуждаются в приложении A.)

Помните! Существуют серьезные проблемы безопасности, связанные с сериализацией и десериализацией. В этом отношении важно ознакомиться с актуальной документацией по Java.

Преимущества потоков

Потоковый интерфейс ввода-вывода в Java обеспечивает чистую абстракцию для сложной и зачастую громоздкой задачи. Состав классов фильтрующих потоков позволяет динамически строить специальный потоковый интерфейс в соответствии с вашими требованиями к передаче данных. Программы на Java, написанные с учетом абстрактных высокоуровневых классов InputStream, OutputStream, Reader и Writer, должны продолжить правильно функционировать в будущем, даже в случае развития конкретных потоковых классов. В главе 24 будет показано, что эта модель работает очень хорошо при переключении с набора потоков, основанных на файловой системе, на сетевые и сокетные потоки. Наконец, сериализация объектов играет важную роль во многих типах программ на Java. Классы ввода-вывода для сериализации в Java предлагают переносимое решение такой временами сложной задачи.

ГЛАВА

23 Исследование системы NIO

Начиная с версии 1.4, в Java предлагается вторая система ввода-вывода под названием *NIO* (New I/O — новый ввод-вывод). Она поддерживает ориентированный на буферы и основанный на каналах подход к выполнению операций ввода-вывода. В выпуске JDK 7 система NIO значительно расширилась, обеспечив усовершенствованную поддержку функциональных средств для обработки файлов и взаимодействия с файловой системой. На самом деле изменения были настолько значительными, что часто применяется термин *NIO.2*. Благодаря возможностям, поддерживаемым файловыми классами NIO, система NIO стала важным подходом к обработке файлов. В настоящей главе рассматриваются некоторые ключевые особенности системы NIO.

Классы NIO

Классы NIO содержатся в пакетах, кратко описанных в табл. 23.1. Начиная с версии JDK 9, все они находятся в модуле `java.base`.

Таблица 23.1. Пакеты с классами NIO

Пакет	Описание
`java.nio`	Классы верхнего уровня системы NIO. Инкапсулирует различные типы буферов, содержащих данные, с которыми работает система NIO
`java.nio.channels`	Поддерживает каналы, которые по существу открывают подключения ввода-вывода
`java.nio.channels.spi`	Поддерживает поставщики служб для каналов
`java.nio.charset`	Инкапсулирует наборы символов. Также поддерживает кодировщики и декодировщики, которые преобразуют соответственно символы в байты и байты в символы

Окончание табл. 23.1

Пакет	Описание
`java.nio.charset.spi`	Поддерживает поставщики служб для наборов символов
`java.nio.file`	Предоставляет поддержку для файлов
`java.nio.file.attribute`	Предоставляет поддержку для файловых атрибутов
`java.nio.file.spi`	Поддерживает поставщики служб для файловой системы

Прежде чем начать, важно подчеркнуть, что система NIO не заменяет классы потокового ввода-вывода из `java.io`, которые обсуждались в главе 22, и хорошее практическое знание потокового ввода-вывода из `java.io` помогает понять работу NIO.

На заметку! В главе предполагается, что вы ознакомились с обзором ввода-вывода, приведенным в главе 13, и обсуждением ввода-вывода на основе потоков, предложенным в главе 22.

Основы NIO

Система NIO построена на двух фундаментальных элементах: буферах и каналах. *Буфер* хранит данные. *Канал* представляет собой открытое подключение к устройству ввода-вывода, такому как файл или сокет. В общем случае для использования системы NIO понадобится получить канал к устройству ввода-вывода и буфер для хранения данных. Затем нужно работать с буфером, по мере необходимости выполняя ввод или вывод данных. В последующих разделах буферы и каналы рассматриваются более подробно.

Буферы

Классы буферов определены в пакете `java.nio`. Все буферы являются подклассами класса `Buffer`, который определяет основную функциональность, общую для всех буферов: текущую позицию, предел и емкость. *Текущая позиция* — это индекс в буфере, в котором будет выполняться следующая операция чтения или записи. Текущая позиция перемещается после выполнения большинства операций чтения или записи. *Предел* представляет собой значение индекса за последним допустимым местоположением в буфере. *Емкость* задает количество элементов, которые способен хранить буфер. Часто предел равен емкости буфера.

Класс `Buffer` также поддерживает пометку и переустановку и определяет методы, кратко описанные в табл. 23.2.

Таблица 23.2. Методы, определенные в классе `Buffer`

Метод	Описание
`abstract Object array()`	Если вызывающий буфер поддерживается массивом, тогда возвращает ссылку на этот массив. В противном случае генерирует исключение `UnsupportedOperationException`. Если массив допускает только чтение, тогда генерирует исключение `ReadOnlyBufferException`
`abstract int arrayOffset()`	Если вызывающий буфер поддерживается массивом, тогда возвращает индекс первого элемента. В противном случае генерирует исключение `UnsupportedOperationException`. Если массив допускает только чтение, тогда генерирует исключение `ReadOnlyBufferException`
`final int capacity()`	Возвращает количество элементов, которые способен хранить вызывающий буфер
`final Buffer clear()`	Очищает вызывающий буфер и возвращает ссылку на него
`abstract Buffer duplicate()`	Возвращает буфер, идентичный вызывающему буферу. Таким образом, оба буфера будут содержать и ссылаться на те же самые элементы
`final Buffer flip()`	Устанавливает предел вызывающего буфера в текущую позицию и сбрасывает текущую позицию в ноль. Возвращает ссылку на буфер
`abstract boolean hasArray()`	Возвращает `true`, если вызывающий буфер поддерживается массивом, допускающим чтение и запись, или `false` в противном случае
`final boolean hasRemaining()`	Возвращает `true`, если в вызывающем буфере остались элементы, или `false` в противном случае
`abstract boolean isDirect()`	Возвращает `true`, если вызывающий буфер является прямым, т.е. операции ввода-вывода воздействуют непосредственно на него. В противном случае возвращает `false`
`abstract boolean isReadOnly()`	Возвращает `true`, если вызывающий буфер допускает только чтение, или `false` в противном случае
`final int limit()`	Возвращает предел вызывающего буфера
`final Buffer limit(int n)`	Устанавливает предел вызывающего буфера в `n`. Возвращает ссылку на буфер

Окончание табл. 23.2

Метод	Описание
`final Buffer mark()`	Устанавливает метку и возвращает ссылку на вызывающий буфер
`final int position()`	Возвращает текущую позицию
`final Buffer position(int n)`	Устанавливает текущую позицию вызывающего буфера в n. Возвращает ссылку на буфер
`int remaining()`	Возвращает количество элементов, доступных в буфере до того, как будет достигнут предел. Другими словами, возвращает разность между пределом и текущей позицией
`final Buffer reset()`	Переустанавливает текущую позицию вызывающего потока в ранее установленную метку. Возвращает ссылку на буфер
`final Buffer rewind()`	Устанавливает позицию вызывающего буфера в ноль. Возвращает ссылку на буфер
`abstract Buffer slice()`	Возвращает буфер, который содержит элементы из вызывающего буфера, начиная с его текущей позиции. Таким образом, оба буфера в срезе будут содержать и ссылаться на те же самые элементы
`abstract Buffer slice (int startIdx, int size)`	Возвращает буфер, который содержит `size` элементов из вызывающего буфера, начиная с позиции `startIdx`. Таким образом, оба буфера в срезе будут содержать и ссылаться на те же самые элементы

От класса `Buffer` унаследованы следующие специальные классы буферов, которые содержат тип данных, вытекающий из их имен:

`ByteBuffer`	`CharBuffer`	`DoubleBuffer`	`FloatBuffer`
`IntBuffer`	`LongBuffer`	`MappedByteBuffer`	`ShortBuffer`

Класс `MappedByteBuffer` является подклассом `ByteBuffer` и применяется для сопоставления файла с буфером.

Все вышеупомянутые буферы предоставляют различные методы `get()` и `put()`, которые позволяют получать данные из буфера или помещать их в буфер. (Разумеется, если буфер доступен только для чтения, то операции `put()` не будут доступны.) В табл. 23.3 кратко описаны методы `get()` и `put()`, определенные в классе `ByteBuffer`. Другие классы буферов имеют аналогичные методы. Все классы буферов также поддерживают методы, выполняющие разнообразные операции с буферами. Например, можно вручную выделить память под буфер с использованием метода `allocate()`, поместить внутрь

буфера массив с применением метода `wrap()`, а также создать подпоследовательность буфера с помощью метода `slice()`.

Таблица 23.3. Методы `get()` и `put()`, определенные в классе `ByteBuffer`

Метод	Описание
`abstract byte get()`	Возвращает байт в текущей позиции
`ByteBuffer get(byte[] vals)`	Копирует вызывающий буфер в массив, на который ссылается `vals`. Возвращает ссылку на буфер. Если в буфере не осталось `vals.length` элементов, тогда генерируется исключение `BufferUnderflowException`
`ByteBuffer get(byte[] vals, int start, int num)`	Копирует `num` элементов из вызывающего буфера в массив, на который ссылается `vals`, начиная с индекса `start`. Возвращает ссылку на буфер. Если в буфере не осталось `num` элементов, тогда генерируется исключение `BufferUnderflowException`
`abstract byte get(int idx)`	Возвращает байт по индексу `idx` внутри вызывающего буфера
`ByteBuffer get(int bufferStartIdx, byte[] vals)`	Копирует все элементы из вызывающего буфера в массив, на который ссылается `vals`, начиная с индекса `bufferStartIdx`. Возвращает ссылку на буфер. Размер массива `vals` должен быть достаточным для сохранения элементов, иначе сгенерируется исключение `IndexOutOfBoundsException`
`ByteBuffer get(int bufferStartIdx, byte[] vals, int arrayStartIdx, int num)`	Копирует `num` элементов из вызывающего буфера, начиная с индекса `bufferStartIdx` буфера, в массив, на который ссылается `vals`, начиная с индекса `arrayStartIdx` массива. Возвращает ссылку на буфер. Размер массива `vals` должен быть достаточным для сохранения элементов, иначе сгенерируется исключение `IndexOutOfBoundsException`
`abstract ByteBuffer put(byte b)`	Копирует `b` в текущую позицию вызывающего буфера. Возвращает ссылку на буфер. Если буфер полон, тогда генерируется исключение `BufferOverflowException`
`final ByteBuffer put(byte[] vals )`	Копирует все элементы из `vals` в вызывающий буфер, начиная с текущей позиции. Возвращает ссылку на буфер. Если сохранить все элементы в буфере невозможно, тогда генерируется исключение `BufferOverflowException`

Окончание табл. 23.3

Метод	Описание
`ByteBuffer put(byte[] vals, int start, int num)`	Копирует num элементов из vals, начиная с позиции start, в вызывающий буфер. Возвращает ссылку на буфер. Если сохранить все элементы в буфере невозможно, тогда генерируется исключение BufferOverflowException
`ByteBuffer put(ByteBuffer bb)`	Копирует элементы из bb в вызывающий буфер, начиная с текущей позиции. Возвращает ссылку на буфер. Если сохранить все элементы в буфере невозможно, тогда генерируется исключение BufferOverflowException
`abstract ByteBuffer put(int idx, byte b)`	Копирует b в позицию idx вызывающего буфера. Возвращает ссылку на буфер
`ByteBuffer put(int bufferStartIdx, byte[] vals)`	Копирует все элементы из массива, на который ссылается vals, в вызывающий буфер, начиная с индекса bufferStartIdx. Возвращает ссылку на буфер
`ByteBuffer put(int bufferStartIdx, byte[] vals, int arrayStartIdx,int num)`	Копирует num элементов из массива, на который ссылается vals, начиная с индекса arrayStartIdx, в вызывающий буфер, начиная с индекса bufferStartIdx. Возвращает ссылку на буфер
`ByteBuffer put(int toBufferStartIdx, ByteBuffer bb, int fromBufferStartIdx, int num)`	Копирует num элементов из буфера, на который ссылается bb, начиная с индекса fromBufferStartIdx, в вызывающий буфер, начиная с индекса toBufferStartIdx. Возвращает ссылку на буфер

Каналы

Классы каналов определены в пакете `java.nio.channels`. Канал представляет собой открытое подключение с источником или адресатом ввода-вывода. Классы каналов реализуют интерфейс Channel, который расширяет интерфейсы Closeable и AutoCloseable. Благодаря реализации AutoCloseable каналами можно управлять с помощью оператора try с ресурсами. При использовании в блоке try с ресурсами канал автоматически закрывается, когда он больше не нужен. (Оператор try с ресурсами обсуждался в главе 13.) Один из способов получения канала предусматривает вызов метода getChannel() на объекте, поддерживающем каналы. Например, метод getChannel() поддерживается следующими классами ввода-вывода:

DatagramSocket	FileInputStream	FileOutputStream
RandomAccessFile	ServerSocket	Socket

Конкретный тип возвращаемого канала зависит от типа объекта, на котором вызывается метод getChannel(). Скажем, при вызове на FileInputStream, FileOutputStream или RandomAccessFile метод getChannel() возвращает канал типа FileChannel, а при вызове на Socket метод getChannel() возвращает SocketChannel.

Еще один способ получения канала предполагает применение одного из статических методов, определенных в классе Files. Например, с использованием Files можно получить байтовый канал, вызвав метод newByteChannel(). Он возвращает объект реализации SeekableByteChannel, который представляет собой интерфейс, реализованный FileChannel. (Класс Files подробно рассматривается далее в главе.)

Каналы вроде FileChannel и SocketChannel поддерживают различные методы read() и write(), позволяющие выполнять операции ввода-вывода через канал. В табл. 23.4 кратко описано несколько методов read() и write(), определенных в классе FileChannel.

Таблица 23.4. Методы read() и write(), определенные в классе FileChannel

Метод	Описание
`abstract int read(ByteBuffer bb) throws IOException`	Читает байты из вызывающего канала в bb, пока буфер не заполнится или не закончатся входные данные. Возвращает количество фактически прочитанных байтов. При попытке чтения в конце файла возвращает -1
`abstract int read(ByteBuffer bb, long start) throws IOException`	Читает байты из вызывающего канала в bb, начиная с позиции start внутри файла, пока буфер не заполнится или не закончатся входные данные. Текущая позиция не изменяется. Возвращает количество фактически прочитанных байтов или -1, если в start указана позиция после конца файла
`abstract int write(ByteBuffer bb) throws IOException`	Записывает содержимое bb в вызывающий канал, начиная с текущей позиции. Возвращает количество записанных байтов
`abstract int write(ByteBuffer bb, long start) throws IOException`	Записывает в вызывающий канал содержимое bb, начиная с позиции start. Текущая позиция не изменяется. Возвращает количество записанных байтов

Все каналы поддерживают дополнительные методы, которые предоставляют доступ к каналу и позволяют им управлять. Скажем, класс FileChannel помимо прочих поддерживает методы для получения или установки текущей позиции, передачи информации между файловыми каналами, получения те-

кущего размера канала и блокировки канала. Класс `FileChannel` предлагает статический метод `open()`, который открывает файл и возвращает канал к нему. Он обеспечивает еще один способ получения канала. Кроме того, класс `FileChannel` предоставляет метод `map()`, предназначенный для сопоставления файла с буфером.

Наборы символов и селекторы

В системе NIO применяются еще две сущности — наборы символов и селекторы. *Набор символов* определяет способ сопоставления байтов с символами. Закодировать последовательность символов в байты можно с помощью кодировщика, а декодировать последовательность байтов в символы — посредством декодировщика. Наборы символов, кодировщики и декодировщики поддерживаются классами, определенными в пакете `java.nio.charset`. По причине наличия стандартных кодировщиков и декодировщиков явно работать с наборами символов придется нечасто.

Селектор поддерживает неблокирующий многоканальный ввод-вывод на основе ключей. Другими словами, селекторы позволяют выполнять ввод-вывод по нескольким каналам. Селекторы поддерживаются классами, определенными в пакете `java.nio.channels`. Они наиболее применимы к каналам, которые поддерживаются сокетами.

В этой главе наборы символов и селекторы не используются, но вы можете счесть их полезными в собственных приложениях.

Усовершенствования, появившиеся в NIO.2

В версии JDK 7 система NIO была существенно расширена и усовершенствована. В дополнение к поддержке оператора `try` с ресурсами (который обеспечивает автоматическое управление ресурсами) улучшения включали три новых пакета (`java.nio.file`, `java.nio.file.attribute` и `java.nio.file.spi`), несколько новых классов, интерфейсов и методов, а также прямую поддержку потокового ввода-вывода. Дополнения значительно расширили возможности применения системы NIO, особенно с файлами. Избранные ключевые дополнения описаны в последующих разделах.

Интерфейс `Path`

Пожалуй, наиболее важным дополнением к системе NIO был интерфейс `Path`, поскольку он инкапсулирует путь к файлу. Как будет показано далее, интерфейс `Path` связывает воедино многие файловые функции NIO.2. Он представляет расположение файла в структуре каталогов, находится в пакете `java.nio.file` и наследует интерфейсы `Watchable`, `Iterable<Path>` и `Comparable<Path>`. Интерфейс `Watchable` описывает объект, изменения которого можно отслеживать. Интерфейсы `Iterable` и `Comparable` рассматривались ранее в книге.

В интерфейсе Path объявлено несколько методов, которые работают с путем. Избранные методы кратко описаны в табл. 23.5. Обратите особое внимание на метод getName(). Он используется для получения элемента пути и работает по индексу. По индексу 0 находится часть пути, ближайшая к корню, который является самым левым элементом пути. Последующие индексы соответствуют элементам справа от корня. Количество элементов в пути можно получить, вызвав метод getNameCount(). Чтобы получить строковое представление всего пути, нужно просто вызвать метод toString(). Обратите внимание на возможность преобразования относительного пути в абсолютный посредством метода resolve().

Таблица 23.5. Избранные методы, определенные в интерфейсе Path

Метод	Описание
`default boolean endsWith (String path)`	Возвращает true, если вызывающий объект реализации Path заканчивается путем, указанным в path, или false в противном случае
`boolean endsWith (Path path)`	Возвращает true, если вызывающий объект реализации Path заканчивается путем, указанным в path, или false в противном случае
`Path getFileName()`	Возвращает имя файла, ассоциированное с вызывающим объектом реализации Path
`Path getName(int idx)`	Возвращает объект реализации Path, который содержит имя элемента пути, указанного с помощью idx, в вызывающем объекте. Крайний слева элемент находится по индексу 0 и является ближайшим к корню. Крайний справа элемент расположен по индексу getNameCount()-1
`int getNameCount()`	Возвращает количество элементов ниже корневого каталога в вызывающем объекте реализации Path
`Path getParent()`	Возвращает объект реализации Path, который содержит полный путь кроме имени файла, указанный вызывающим объектом реализации Path
`Path getRoot()`	Возвращает корень вызывающего объекта реализации Path
`boolean isAbsolute()`	Возвращает true, если вызывающий объект реализации Path соответствует абсолютному пути, или false в противном случае

Окончание табл. 23.5

Метод	Описание
`static Path of(String pathname, String ... parts)`	Возвращает объект реализации Path, который инкапсулирует указанный путь. Если параметр parts не используется, тогда путь должен быть указан полностью в pathname, иначе к pathname добавляются аргументы, переданные через parts (обычно с надлежащим разделителем), чтобы сформировать полный путь. В любом случае, если указанный путь синтаксически некорректен, тогда сгенерируется исключение InvalidPathException
`static Path of(URI uri)`	Возвращает путь, соответствующий uri
`Path resolve (Path path)`	Если путь path является абсолютным, тогда возвращается path. Иначе, если путь path не содержит корень, то path дополняется префиксом в виде корня, указанного в вызывающем объекте реализации Path, и результат возвращается. Если путь path пуст, тогда возвращается вызывающий объект реализации Path. В противном случае поведение не определено
`default Path resolve (String path)`	Если путь path является абсолютным, тогда возвращается path. Иначе, если путь path не содержит корень, то path дополняется префиксом в виде корня, указанного в вызывающем объекте реализации Path, и результат возвращается. Если путь path пуст, тогда возвращается вызывающий объект реализации Path. В противном случае поведение не определено
`default boolean startsWith (String path)`	Возвращает true, если путь в вызывающем объекте реализации Path начинается с пути, указанного в path, или false в противном случае
`boolean startsWith (Path path)`	Возвращает true, если путь в вызывающем объекте реализации Path начинается с пути, указанного в path, или false в противном случае
`Path toAbsolutePath()`	Возвращает вызывающий объект реализации Path как абсолютный путь
`String toString()`	Возвращает строковое представление вызывающего объекта реализации Path

Начиная с версии JDK 11, в интерфейс Path был добавлен новый важный статический фабричный метод по имени of(). Он возвращает экземпляр реализации Path, созданный либо из имени пути, либо из URI. Таким образом, метод of() дает возможность конструирования нового экземпляра реализации Path.

И еще один момент: при обновлении унаследованного кода, где задействован класс File, определенный в java.io, экземпляр File можно преобразовать в экземпляр реализации Path, вызвав метод toPath() на объекте File. Кроме того, экземпляр File можно получить, вызвав метод toFile(), который определен в интерфейсе Path.

Класс Files

Многие действия, выполняемые с файлом, обеспечиваются статическими методами класса Files. Файл, с которым нужно работать, определяется его объектом реализации Path, т.е. методы класса Files применяют объект реализации Path для указания файла, подвергающегося операции. Класс Files предлагает широкий спектр функциональных средств. Например, у него есть методы, позволяющие открыть или создать файл с указанным путем. Можно получить информацию об объекте реализации Path, касающуюся того, является он исполняемым, скрытым или допускающим только чтение. Класс Files также предоставляет методы, предназначенные для копирования или перемещения файлов. В табл. 23.6 кратко описаны избранные методы, определенные в Files. Помимо IOException могут возникать и другие исключения. Кроме того, класс Files содержит следующие четыре метода: list(), walk(), lines() и find(). Все они возвращают объект Stream и помогают интегрировать систему NIO с потоковым API, рассматриваемым в главе 30. В состав Files также входят методы readString() и writeString(), которые возвращают строку, содержащую символы в файле, или записывают в файл объект типа CharSequence (например, String).

Таблица 23.6. Избранные методы, определенные в классе Files

Метод	Описание
`static Path copy (Path src, Path dest, CopyOption ... how) throws IOException`	Копирует файл, указанный с помощью src, в местоположение, заданное dest. В параметре how указывается, каким образом будет происходить копирование
`static Path createDirectory (Path path, FileAttribute<?>... attribs) throws IOException`	Создает каталог, путь к которому указан в path. Атрибуты каталога задаются в attribs
`static Path createFile(Path path, FileAttribute<?> ... attribs) throws IOException`	Создает файл, путь к которому указан в path. Атрибуты файла задаются в attribs

Продолжение табл. 23.6

Метод	Описание
`static void delete(Path path) throws IOException`	Удаляет файл, путь к которому указан в path
`static boolean exists(Path path, LinkOption ... opts)`	Возвращает true, если файл, заданный посредством path, существует, или false в противном случае. Если параметр opts не указан, тогда происходит переход по символическим ссылкам. Чтобы предотвратить переходы по символическим ссылкам, в opts потребуется передать NOFOLLOW_LINKS
`static boolean isDirectory(Path path, LinkOption ... opts)`	Возвращает true, если в path указан каталог, или false в противном случае. Если параметр opts не указан, тогда происходит переход по символическим ссылкам. Чтобы предотвратить переходы по символическим ссылкам, в opts потребуется передать NOFOLLOW_LINKS
`static boolean isExecutable(Path path)`	Возвращает true, если файл, заданный посредством path, является исполняемым, или false в противном случае
`static boolean isHidden(Path path) throws IOException`	Возвращает true, если файл, заданный посредством path, является скрытым, или false в противном случае
`static boolean isReadable(Path path)`	Возвращает true, если файл, заданный посредством path, допускает чтение, или false в противном случае
`static boolean isRegularFile(Path path, LinkOption ... opts)`	Возвращает true, если в path указан файл, или false в противном случае. Если параметр opts не задан, тогда происходит переход по символическим ссылкам. Чтобы предотвратить переходы по символическим ссылкам, в opts потребуется передать NOFOLLOW_LINKS
`static boolean isWritable(Path path)`	Возвращает true, если файл, заданный посредством path, допускает запись, или false в противном случае
`static Path move(Path src, Path dest, CopyOption ... how) throws IOException`	Перемещает файл, указанный в src, в местоположение, заданное в dest. В параметре how указывается, каким образом будет происходить перемещение

Окончание табл. 23.6

Метод	Описание
`static SeekableByteChannel newByteChannel(Path path, OpenOption ... how) throws IOException`	Открывает файл, заданный посредством path, как указано в how. Возвращает для файла объект реализации SeekableByteChannel, который является байтовым каналом с текущей позицией, допускающей изменение. Интерфейс SeekableByteChannel реализован классом FileChannel
`static DirectoryStream<Path> newDirectoryStream(Path path) throws IOException`	Открывает каталог, указанный в path. Возвращает объект реализации DirectoryStream, связанный с каталогом
`static InputStream newInputStream(Path path, OpenOption ... how) throws IOException`	Открывает файл, заданный посредством path, как указано в how. Возвращает объект InputStream, связанный с файлом
`static OutputStream newOutputStream(Path path, OpenOption ... how) throws IOException`	Открывает файл, заданный посредством вызывающего объекта, как указано в how. Возвращает объект OutputStream, связанный с файлом
`static boolean notExists(Path path, LinkOption ... opts)`	Возвращает true, если файл, заданный посредством path, не существует, или false в противном случае. Если параметр opts не указан, тогда происходит переход по символическим ссылкам. Чтобы предотвратить переходы по символическим ссылкам, в opts потребуется передать NOFOLLOW_LINKS
`static <A extends BasicFileAttributes> A readAttributes(Path path, Class<A> attribType, LinkOption ... opts) throws IOException`	Получает атрибуты, ассоциированные с файлом, который указан в path. В attribType передается тип получаемых атрибутов. Если параметр opts не указан, тогда происходит переход по символическим ссылкам. Чтобы предотвратить переходы по символическим ссылкам, в opts потребуется передать NOFOLLOW_LINKS
`static long size(Path path) throws IOException`	Возвращает размер файла, указанного с помощью path

Обратите внимание, что несколько методов в табл. 23.6 принимают аргумент типа OpenOption. Это интерфейс, который описывает, каким образом должен быть открыт файл. Он реализуется перечислением StandardOpenOption со значениями, показанными в табл. 23.7.

Таблица 23.7. Значения перечисления StandardOpenOption

Значение	Описание
APPEND	Обеспечить запись выходных данных в конец файла
CREATE	Создать файл, если он не существует
CREATE_NEW	Создать файл только в случае, если он не существует
DELETE_ON_CLOSE	Удалить файл после его закрытия
DSYNC	Обеспечить немедленную запись изменений файла в физический файл. Обычно изменения, вносимые в файл, буферизуются файловой системой в интересах эффективности и записываются в файл только по мере необходимости
READ	Открыть файл для операций ввода
SPARSE	Указать файловой системе о том, что файл является разреженным, т.е. он может быть заполнен данными не полностью. Если файловая система не поддерживает разреженные файлы, то этот параметр игнорируется
SYNC	Обеспечить немедленную запись изменений файла или его метаданных в физический файл. Обычно изменения, вносимые в файл, буферизуются файловой системой в интересах эффективности и записываются в файл только по мере необходимости
TRUNCATE_EXISTING	Обеспечить обнуление длины существующего файла, открытого для вывода
WRITE	Открыть файл для операций вывода

Класс Paths

Поскольку Path является интерфейсом, а не классом, создавать экземпляр Path напрямую с помощью конструктора нельзя. Взамен нужно получить экземпляр реализации Path, вызывая метод, который его возвращает. До JDK 11 это обычно делалось с помощью метода get(), определенного в классе Paths. Существуют две формы get(). Вот первая форма:

```
static Path get(String pathname, String ... parts)
```

Первая форма метода get() возвращает объект реализации Path, который инкапсулирует указанный путь. Путь можно указывать двумя способами.

Если параметр parts не используется, тогда путь должен быть указан целиком в pathname. В качестве альтернативы путь можно указывать по частям, причем первая часть передается в pathname, а последующие элементы — в параметре parts. В любом случае, если указанный путь синтаксически некорректен, метод get() сгенерирует исключение InvalidPathException. Вторая форма метода get() создает объект реализации Path из объекта URI:

```
static Path get(URI uri)
```

Она возвращает объект реализации Path, соответствующий uri.

Хотя только что описанный метод Paths.get() применяется, начиная с версии JDK 7, и на момент написания главы все еще доступен для использования, он больше не является рекомендуемым. В документации по Java API теперь рекомендуется применять новый метод Path.of(), который появился в JDK 11 и считается более предпочтительным. Конечно, если вы имеете дело с версией компилятора, предшествующей JDK 11, то должны продолжать пользоваться Paths.get().

Важно понимать, что получение объекта реализации Path для файла не приводит к открытию или созданию файла. Просто в итоге получается объект, который инкапсулирует путь к каталогу файла.

Интерфейсы для файловых атрибутов

С файлом ассоциирован набор атрибутов. К ним относятся такие данные, как время создания файла, время его последней модификации, признак, является ли файл каталогом, и его размер. Система NIO организует атрибуты файлов в несколько разных интерфейсов. Атрибуты представлены иерархией интерфейсов, определенных в java.nio.file.attribute. На вершине иерархии находится интерфейс BasicFileAttributes, инкапсулирующий набор атрибутов, которые обычно встречаются в разнообразных файловых системах. Методы, определенные в BasicFileAttributes, кратко описаны в табл. 23.8.

Таблица 23.8. Методы, определенные в интерфейсе BasicFileAttributes

Метод	Описание
FileTime creationTime()	Возвращает время создания файла. Если файловая система не предоставляет время создания, тогда возвращается значение, зависящее от реализации
Object fileKey()	Возвращает файловый ключ. Если он не поддерживается, тогда возвращается null
boolean isDirectory()	Возвращает true, если файл представляет собой каталог
boolean isOther()	Возвращает true, если файл не является файлом, символической ссылкой или каталогом

Окончание табл. 23.8

Метод	Описание
`boolean isRegularFile()`	Возвращает `true`, если файл является обычным файлом, а не каталогом или символической ссылкой
`boolean isSymbolicLink()`	Возвращает `true`, если файл является символической ссылкой
`FileTime lastAccessTime()`	Возвращает время последнего доступа к файлу. Если файловая система не предоставляет время последнего доступа, тогда возвращается значение, зависящее от реализации
`FileTime lastModifiedTime()`	Возвращает время последней модификации файла. Если файловая система не предоставляет время последней модификации, тогда возвращается значение, зависящее от реализации
`long size()`	Возвращает размер файла

Два интерфейса являются производными от `BasicFileAttributes`: `DosFileAttributes` и `PosixFileAttributes`. Интерфейс `DosFileAttributes` описывает атрибуты, относящиеся к файловой системе FAT, как они были впервые определены в операционной системе DOS. В нем определены методы, показанные в табл. 23.9.

Таблица 23.9. Методы, определенные в интерфейсе `DosFileAttributes`

Метод	Описание
`boolean isArchive()`	Возвращает `true`, если файл помечен как архивный, или `false` в противном случае
`boolean isHidden()`	Возвращает `true`, если файл является скрытым, или `false` в противном случае
`boolean isReadOnly()`	Возвращает `true`, если файл допускает только чтение, или `false` в противном случае
`boolean isSystem()`	Возвращает `true`, если файл помечен как системный, или `false` в противном случае

Интерфейс `PosixFileAttributes` инкапсулирует атрибуты, определенные стандартами POSIX (Portable Operating System Interface — интерфейс переносимой операционной системы). Он определяет методы, кратко описанные в табл. 23.10.

Таблица 23.10. Методы, определенные в интерфейсе `PosixFileAttributes`

Метод	Описание
`GroupPrincipal group()`	Возвращает группового владельца файла
`UserPrincipal owner()`	Возвращает владельца файла
`Set<PosixFilePermission> permissions()`	Возвращает разрешения файла

Существуют различные способы доступа к атрибутам файла. Для начала получить объект, который инкапсулирует атрибуты файла, можно посредством вызова `readAttributes()` — статического метода, определенного в `Files`. Вот одна из его форм:

```
static <A extends BasicFileAttributes>
  A readAttributes(Path path, Class<A> attrType, LinkOption ... opts)
    throws IOException
```

Метод `readAttributes()` возвращает ссылку на объект, указывающий атрибуты, которые ассоциированы с файлом, переданным в `path`. Специфический тип атрибутов задается как объект `Class` в параметре `attrType`. Скажем, для получения базовых атрибутов файла в `attrType` необходимо передать `BasicFileAttributes.class`. Для атрибутов DOS должно передаваться `DosFileAttributes.class`, а для атрибутов POSIX — `PosixFileAttributes.class`. Необязательные параметры ссылок передаются в `opts`. Если `opts` не применяется, тогда происходит следование по символическим ссылкам. Метод возвращает ссылку на запрошенные атрибуты. Если запрошенный тип атрибута недоступен, тогда генерируется исключение `UnsupportedOperationException`. С помощью возвращенного объекта можно получить доступ к атрибутам файла.

Второй способ получения доступа к атрибутам файла предусматривает вызов метода `getFileAttributeView()`, определенного в `Files`. В системе NIO определено несколько интерфейсов представлений атрибутов, включающие среди прочих `AttributeView`, `BasicFileAttributeView`, `DosFileAttributeView` и `PosixFileAttributeView`. Хотя в этой главе представления атрибутов использоваться не будут, в некоторых ситуациях они могут оказаться полезными.

В ряде случаев интерфейсы файловых атрибутов напрямую задействовать не придется, потому что класс `Files` предлагает удобные статические методы, которые открывают доступ к нескольким атрибутам. Например, в классе `Files` определены такие методы, как `isHidden()` и `isWritable()`.

Важно понимать, что не каждая файловая система поддерживает абсолютно все возможные атрибуты. Скажем, атрибуты файлов DOS относятся к более старой файловой системе FAT, как она изначально была определена в DOS. Атрибуты, которые будут применяться к широкому разнообразию файловых систем, описаны с помощью интерфейса `BasicFileAttributes` и потому они используются в примерах, рассматриваемых в текущей главе.

Классы FileSystem, FileSystems и FileStore

Получить доступ к файловой системе очень легко посредством классов `FileSystem` и `FileSystems` из пакета `java.nio.file`. На самом деле с применением метода `newFileSystem()`, определенного в `FileSystems`, можно даже получить новую файловую систему. Класс `FileStore` инкапсулирует систему хранения файлов. Хотя упомянутые классы в этой главе напрямую не используются, они могут оказаться полезными в реальных приложениях.

Использование системы NIO

В настоящем разделе будет показано, как применять систему NIO для решения разнообразных задач. Прежде чем начать, важно подчеркнуть, что в JDK 7 система NIO была существенно расширена, в результате чего границы ее использования тоже значительно расширились. Как уже упоминалось, расширенную версию иногда называют NIO.2. Поскольку функциональные средства, добавленные NIO.2, настолько значимы, они изменили способ написания большей части кода на основе NIO и увеличили количество типов задач, для решения которых можно применять NIO. По причине своей важности в приводимом далее обсуждении и примерах используются средства NIO.2, так что для них требуется современная версия Java.

В прошлом главным предназначением системы NIO был ввод-вывод, основанный на каналах, который все еще остается очень важным применением. Однако теперь систему NIO можно использовать для потокового ввода-вывода и выполнения операций с файловой системой. В результате обсуждение применения системы NIO разделено на три части:

- использование NIO для ввода-вывода, основанного на каналах;
- применение NIO для ввода-вывода, основанного на потоках данных;
- использование NIO для выполнения операций с путями и файловой системой.

Поскольку наиболее распространенным устройством ввода-вывода является дисковый файл, такие файлы будут применяться в примерах, приведенных в оставшемся материале главы. Так как все операции файлового канала основаны на байтах, будут использоваться буферы типа `ByteBuffer`.

Прежде чем появится возможность открыть файл для доступа через систему NIO, необходимо получить объект реализации `Path`, описывающий файл. В прошлом один из способов достижения такой цели предполагал вызов фабричного метода `Paths.get()`, но, как объяснялось ранее, в версии JDK 11 предпочтительным подходом стало применение метода `Path.of()`, а не `Paths.get()`, так что он и будет использоваться в примерах. В случае работы с версией Java, предшествующей JDK 11, в программах понадобится просто заменить вызов `Path.of()` вызовом `Path.get()`. Ниже показана форма метода `of()`, применяемая в примерах:

```
static Path of(String pathname, String ... parts)
```

Вспомните, что путь можно указывать двумя способами. Его можно передавать по частям, причем первая часть передается в pathname, а последующие элементы указываются в параметре parts. В качестве альтернативы в pathname можно задать полный путь, а parts не использовать. Именно такой подход задействован в примерах.

Использование системы NIO для ввода-вывода, основанного на каналах

Важным применением системы NIO является доступ к файлу через канал и буферы. В последующих разделах демонстрируются некоторые приемы использования канала для чтения и записи в файл.

Чтение файла через канал

Существует несколько приемов чтения данных из файла с применением канала. Возможно, наиболее распространенным способом считается выделение памяти под буфер вручную, а затем выполнение явной операции чтения, которая загружает этот буфер данными из файла. Начнем с исследования такого подхода.

Прежде чем можно будет читать из файла, его потребуется открыть, для чего сначала создать объект реализации Path, описывающий файл. Затем готовый объект реализации Path используется при открытии файла. Есть разные способы открытия файла, которые зависят от того, как он будет эксплуатироваться. В рассматриваемом примере файл будет открыт для байтового ввода посредством явных операций ввода. Таким образом, в примере открывается файл, для которого с помощью вызова Files.newByteChannel() устанавливается канал. Применяемая версия метода newByteChannel() имеет следующую общую форму:

```
static SeekableByteChannel newByteChannel(Path path, OpenOption ... how)
  throws IOException
```

Метод newByteChannel() возвращает объект реализации SeekableByteChannel, который инкапсулирует канал для файловых операций. В параметре path передается объект реализации Path, описывающий файл. В параметре how задается способ открытия файла. Поскольку how — аргумент переменной длины, можно указывать ноль или большее число значений, разделяя их запятыми. (Допустимые значения обсуждались ранее и показаны в табл. 23.7.) Если аргументы отсутствуют, тогда файл открывается для операций ввода. SeekableByteChannel представляет собой интерфейс, описывающий канал, который можно использовать для файловых операций. Он реализован классом FileChannel. Когда применяется стандартная файловая система, возвращаемый объект может быть приведен к FileChannel. После завершения работы канал должен быть закрыт. Поскольку все каналы, включая FileChannel, реализуют интерфейс AutoCloseable, вместо явного вызова метода close() можно использовать оператор try с ресурсами для автоматического закрытия файла. Такой подход задействован в примерах.

Затем необходимо получить буфер, который будет применяться каналом, либо за счет создания оболочки вокруг существующего массива, либо путем динамического выделения памяти под буфер. В примерах используется выделение памяти, но выбор за вами. Из-за того, что файловые каналы работают с байтовыми буферами, для получения буфера будет применяться метод `allocate()`, определенный в `ByteBuffer`. Вот его общий вид:

```
static ByteBuffer allocate(int cap)
```

Здесь в `cap` указывается емкость буфера. Метод `allocate()` возвращает ссылку на буфер.

После создания буфера понадобится вызвать метод `read()` на канале, передав ему ссылку на буфер. Ниже показана используемая версия `read()`:

```
int read(ByteBuffer buf) throws IOException
```

При каждом вызове метод `read()` заполняет буфер, указанный в `buf`, данными из файла. Операции чтения выполняются последовательно, т.е. каждый вызов `read()` читает из файла байты для следующего буфера. Метод `read()` возвращает количество фактически прочитанных байтов или `-1` при попытке чтения в конце файла.

Предыдущее обсуждение воплощено в следующей программе, в которой читается файл по имени `test.txt` через канал с применением явных операций ввода:

```
// Использование канального ввода-вывода для чтения файла.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelRead {
  public static void main(String[] args) {
    int count;
    Path filepath = null;

    // Сначала получить путь к файлу.
    try {
      filepath = Path.of("test.txt");
    } catch(InvalidPathException e) {
      System.out.println("Ошибка в пути: " + e);
      return;
    }

    // Затем получить канал к этому файлу внутри блока try с ресурсами.
    try ( SeekableByteChannel fChan = Files.newByteChannel(filepath) )
    {
      // Выделить память под буфер.
      ByteBuffer mBuf = ByteBuffer.allocate(128);

      do {
        // Читать данные в буфер.
        count = fChan.read(mBuf);
```

```
        // Остановиться, когда достигнут конец файла.
        if(count != -1) {
          // Переустановить буфер в начало, чтобы из него можно было читать.
          mBuf.rewind();
          // Прочитать байты из буфера и отобразить их на экране как символы.
          for(int i=0; i < count; i++)
            System.out.print((char)mBuf.get());
        }
      } while(count != -1);
      System.out.println();
    } catch (IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

Ниже описана работа программы. Сначала получается объект реализации Path, содержащий относительный путь к файлу по имени test.txt. Ссылка на этот объект присваивается filepath. Затем получается канал, подключенный к файлу, посредством вызова newByteChannel() с передачей ему filepath. Поскольку параметры открытия не указаны, файл открывается для чтения. Обратите внимание, что канал является объектом, который управляется оператором try с ресурсами. Таким образом, когда блок заканчивается, канал автоматически закрывается. Далее программа вызывает метод allocate() объекта ByteBuffer для выделения памяти под буфер, где будет сохраняться содержимое файла при его чтении. Ссылка на буфер помещается в mBuf. Содержимое файла читается в mBuf по одному буферу за раз с помощью вызова read(). Количество прочитанных байтов хранится в переменной count. Буфер переустанавливается в начало вызовом rewind(), который необходим из-за того, что после вызова read() текущая позиция находится в конце буфера. Она должна быть сброшена в начало буфера, чтобы байты можно было читать из mBuf, вызывая get(). (Вспомните, что метод get() определен в классе ByteBuffer.) Так как mBuf ссылается на байтовый буфер, возвращаемые методом get() значения являются байтами. Они приводятся к типу char, так что файл может быть отображен в виде текста. (В качестве альтернативы можно создать буфер, который кодирует байты в символы, и затем читать этот буфер.) В случае достижения конца файла метод read() возвращает -1, программа завершается, а канал автоматически закрывается.

Интересно отметить, что программа получает объект реализации Path в одном блоке try, после чего использует другой блок try для получения и управления каналом, связанным с этим объектом Path. Хотя в таком подходе нет ничего плохого, во многих случаях его можно упростить так, чтобы требовался только один блок try, упорядочив вместе вызовы Path.of() и newByteChannel(). Вот переработанная версия программы:

```
// Более компактный способ открытия канала.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
```

```
public class ExplicitChannelRead {
  public static void main(String[] args) {
    int count;

    // Здесь канал открывается на объекте реализации Path,
    // возвращенном методом Path.of().
    // Нет необходимости в переменной filepath.
    try ( SeekableByteChannel fChan =
          Files.newByteChannel(Path.of("test.txt")) )
    {
      // Выделить память под буфер.
      ByteBuffer mBuf = ByteBuffer.allocate(128);

      do {
        // Читать данные в буфер.
        count = fChan.read(mBuf);

        // Остановиться, когда достигнут конец файла.
        if(count != -1) {

          // Переустановить буфер в начало, чтобы из него можно было читать.
          mBuf.rewind();

          // Прочитать байты из буфера и отобразить их на экране как символы.
          for(int i=0; i < count; i++)
            System.out.print((char)mBuf.get());
        }
      } while(count != -1);

      System.out.println();
    } catch(InvalidPathException e) {
      System.out.println("Ошибка в пути: " + e);
    } catch (IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

В новой версии переменная `filepath` не нужна, к тому же оба исключения обрабатываются одним оператором `try`. Поскольку такой подход более компактен, именно он применяется в остальных примерах, рассмотренных в главе. Конечно, в реальном коде могут возникать ситуации, когда создание объекта реализации `Path` должно быть отделено от получения канала; тогда можно использовать предыдущий подход.

Другой способ чтения файла предусматривает его сопоставление с буфером. Преимущество такого подхода в том, что буфер автоматически содержит данные из файла. Отпадает необходимость в явной операции чтения. Чтобы сопоставить и прочитать содержимое файла, потребуется выполнить описанную далее общую процедуру. Сначала понадобится получить объект реализации `Path`, который инкапсулирует файл, как упоминалось ранее. Затем нужно получить канал для файла посредством вызова метода `Files.newByteChannel()`, передав ему объект `Path`, и привести возвращаемый объект к типу `FileChannel`. Как уже объяснялось, метод `newByteChannel()` возвращает объект реализации `SeekableByteChannel`. При работе со стан-

дартной файловой системой его можно привести к FileChannel. Наконец, необходимо сопоставить канал с буфером, вызвав map() на канале. Метод map() определен в FileChannel — вот почему приведение к FileChannel обязательно. Метод map() выглядит следующим образом:

```
MappedByteBuffer map(FileChannel.MapMode how,
  long pos, long size) throws IOException
```

Метод map() сопоставляет данные в файле с буфером в памяти. Значение в параметре how определяет, какие виды операций разрешены. Допустимые значения перечислены ниже:

MapMode.READ_ONLY MapMode.READ_WRITE MapMode.PRIVATE

Для чтения файла следует применять MapMode.READ_ONLY, а для чтения и записи — MapMode.READ_WRITE. Значение MapMode.PRIVATE обеспечивает создание закрытой копии файла, так что изменения в буфере не влияют на лежащий в основе файл. Местоположение в файле, с которого начинается сопоставление, указывается в pos, а количество байтов для сопоставления задается в size. Ссылка на буфер возвращается как объект класса MappedByteBuffer, который является подклассом ByteBuffer. После сопоставления файла с буфером файл можно читать из этого буфера. Вот пример, иллюстрирующий такой подход:

```
// Использование сопоставления файла с буфером для чтения файла.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class MappedChannelRead {
  public static void main(String[] args) {

    // Получить канал к файлу внутри блока try с ресурсами.
    try ( FileChannel fChan =
         (FileChannel) Files.newByteChannel(Path.of("test.txt")) )
    {

      // Получить размер файла.
      long fSize = fChan.size();

      // Сопоставить файл с буфером.
      MappedByteBuffer mBuf =
        fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

      // Прочитать байты из буфера и отобразить их на экране как символы.
      for(int i=0; i < fSize; i++)
        System.out.print((char)mBuf.get());
      System.out.println();
    } catch(InvalidPathException e) {
      System.out.println("Ошибка в пути: " + e);
    } catch (IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

В программе создается объект реализации Path для файла, который открывается с помощью newByteChannel(). Объект канала приводится к типу FileChannel и сохраняется в fChan. Затем через вызов size() на канале получается размер файла. Далее весь файл сопоставляется с буфером в памяти посредством вызова метода map() на fChan, а ссылка на буфер сохраняется в mBuf. Обратите внимание, что mBuf объявляется как ссылка на MappedByteBuffer. Байты читаются из mBuf вызовом get().

Запись в файл через канал

Как и в случае чтения из файла, существует несколько способов записи данных в файл с использованием канала. Для начала рассмотрим один из наиболее распространенных подходов. Он предусматривает выделение памяти под буфер вручную, запись данных в этот буфер и выполнение явной операции записи для помещения данных в файл.

Прежде чем можно будет записывать данные в файл, его потребуется открыть, сначала получив объект реализации Path, описывающий файл, а затем применив этот объект Path для открытия файла. В настоящем примере файл будет открыт для байтового вывода через явные операции вывода. Следовательно, код в примере откроет файл и установит для него канал, вызвав Files.newByteChannel(). Как показано в предыдущем разделе, метод newByteChannel(), который будет использоваться, имеет такую общую форму:

```
static SeekableByteChannel newByteChannel(Path path, OpenOption ... how)
  throws IOException
```

Метод newByteChannel() возвращает объект реализации SeekableByteChannel, который инкапсулирует канал для файловых операций. Чтобы открыть файл для вывода, в how должно быть указано StandardOpenOption.WRITE. Если необходимо создать файл, если он не существует, то понадобится также указать StandardOpenOption.CREATE. (Доступны и другие параметры, перечисленные в табл. 23.7.) Как объяснялось в предыдущем разделе, SeekableByteChannel представляет собой интерфейс, описывающий канал, который можно применять для операций с файлами. Он реализован классом FileChannel. Когда используется стандартная файловая система, возвращаемый объект может быть приведен к типу FileChannel. После окончания работы с каналом он должен быть закрыт.

Рассмотрим способ записи в файл через канал, при котором применяются явные вызовы метода write(). Сначала необходимо получить путь к файлу, после чего открыть его с помощью вызова newByteChannel() и привести результат к типу FileChannel. Затем понадобится выделить байтовый буфер и записать в него данные. Перед записью данных в файл нужно вызвать метод rewind() на буфере, чтобы установить его текущую позицию в ноль. (Каждая операция вывода в буфер перемещает текущую позицию вперед, а потому до выполнения записи в файл она должна быть переустановлена.) Наконец, для канала потребуется вызвать метод write(), передав ему буфер. В следующей программе демонстрируется описанная процедура на примере записи алфавита в файл по имени test.txt.

```
// Запись в файл с использованием системы NIO.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelWrite {
  public static void main(String[] args) {

    // Получить канал к файлу внутри блока try с ресурсами.
    try ( FileChannel fChan = (FileChannel)
          Files.newByteChannel(Path.of("test.txt"),
          StandardOpenOption.WRITE, StandardOpenOption.CREATE) )
    {
      // Выделить память под буфер.
      ByteBuffer mBuf = ByteBuffer.allocate(26);

      // Записать несколько байтов в буфер.
      for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));

      // Переустановить буфер, чтобы его можно было записать в файл.
      mBuf.rewind();

      // Записать буфер в выходной файл.
      fChan.write(mBuf);
    } catch(InvalidPathException e) {
      System.out.println("Ошибка в пути: " + e);
    } catch (IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
      System.exit(1);
    }
  }
}
```

Полезно подчеркнуть один важный аспект приведенной программы. Как уже упоминалось, после записи данных в mBuf, но до того, как они будут записаны в файл, выполняется вызов `rewind()` на mBuf. Он необходим для переустановки текущей позиции в ноль после записи данных в mBuf. Не забывайте, что каждый вызов `put()` на mBuf перемещает текущую позицию вперед. Поэтому перед вызовом `write()` текущая позиция должна быть переустановлена в начало буфера, иначе метод `write()` посчитает, что данные в буфере отсутствуют.

Другой способ справиться с переустановкой буфера между операциями ввода и вывода предусматривает вызов `flip()` вместо `rewind()`. Метод `flip()` устанавливает текущую позицию в ноль, а предел — в предыдущую текущую позицию. В показанном выше примере из-за того, что емкость буфера была равна его пределу, вместо `rewind()` можно было использовать метод `flip()`. Тем не менее, не во всех случаях указанные два метода взаимозаменяемы.

Обычно буфер должен быть переустановлен между операциями чтения и записи. Скажем, продолжая предыдущий пример, в показанном ниже цикле алфавит записывается в файл три раза. Обратите особое внимание на вызовы `rewind()`.

```
for(int h=0; h<3; h++) {
  // Записать несколько байтов в буфер.
  for(int i=0; i<26; i++)
    mBuf.put((byte)('A' + i));

  // Переустановить буфер, чтобы его можно было записать в файл.
  mBuf.rewind();

  // Записать буфер в выходной файл.
  fChan.write(mBuf);

  // Переустановить буфер, чтобы его можно было снова записать в файл.
  mBuf.rewind();
}
```

Как видите, метод `rewind()` вызывается между каждой операцией чтения и записи.

Заслуживает упоминания еще одна особенность программы: когда буфер записывается в файл, выходные данные будут содержаться в первых 26 байтах файла. Если файл `test.txt` уже существует, то после выполнения программы алфавит будет храниться в первых 26 байтах файла, а остальная часть файла не изменится.

Другой способ записи в файл предполагает его сопоставление с буфером. Преимущество такого подхода в том, что данные, записываемые в буфер, автоматически записываются и в файл. Никакой явной операции записи не требуется. Для сопоставления и записи содержимого файла будет применяться описанная далее общая процедура. Сначала нужно получить объект реализации `Path`, инкапсулирующий файл, после чего создать канал для этого файла, вызвав метод `Files.newByteChannel()` и передав ему объект `Path`. Затем необходимо привести ссылку, возвращенную методом `newByteChannel()`, к типу `FileChannel`, и сопоставить канал с буфером с помощью вызова `map()` на канале. Метод `map()` был подробно описан в предыдущем разделе. Вспомните его общий вид:

```
MappedByteBuffer map(FileChannel.MapMode how,
  long pos, long size) throws IOException
```

Метод `map()` обеспечивает сопоставление данных в файле с буфером в памяти. Значение в параметре `how` определяет, какие типы операций разрешены. Для записи в файл в `how` должно быть передаваться `MapMode.READ_WRITE`. Местоположение в файле, с которого начинается сопоставление, указывается в `pos`, а количество байтов для сопоставления — в `size`. Метод `map()` возвращает ссылку на буфер. После сопоставления файла с буфером можно записывать данные в буфер, и они будут автоматически записываться в файл, не требуя никаких явных операций записи в канал.

Далее представлена версия предыдущей программы, в которой используется сопоставленный файл. Обратите внимание, что в вызов `newByteChannel()` добавлен параметр открытия `StandardOpenOption.READ`, наличие которого связано с тем, что отображаемый буфер может допускать либо только чтение, либо чтение и запись. Таким образом, для записи в сопоставленный буфер канал должен быть открыт для чтения и записи.

```
// Запись в сопоставленный файл.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class MappedChannelWrite {
  public static void main(String[] args) {

    // Получить канал к файлу внутри блока try с ресурсами.
    try ( FileChannel fChan = (FileChannel)
          Files.newByteChannel(Path.of("test.txt"),
          StandardOpenOption.WRITE, StandardOpenOption.READ,
             StandardOpenOption.CREATE) )
    {
      // Сопоставить файл с буфером.
      MappedByteBuffer mBuf =
        fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);

      // Записать несколько байтов в буфер.
      for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));
    } catch(InvalidPathException e) {
      System.out.println("Ошибка в пути: " + e);
    } catch (IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

Несложно заметить, что явные операции записи канале отсутствуют. Поскольку буфер `mBuf` сопоставляется с файлом, изменения в `mBuf` автоматически отражаются внутри лежащего в основе файла.

Копирование файла с использованием системы NIO

Система NIO упрощает несколько видов операций с файлами. Хотя исследовать здесь абсолютно все операции невозможно, рассмотрим пример, который даст представление о том, что доступно. В следующей программе копируется файл с применением вызова единственного метода NIO — `copy()`, который является статическим методом, определенным в классе `Files`. Он имеет несколько форм и вот та, которая будет использоваться:

```
static Path copy(Path src, Path dest, CopyOption ... how)
   throws IOException
```

Файл, указанный в `src`, копируется в файл, заданный в `dest`. Параметр `how` определяет, каким образом выполняется копирование. Поскольку `how` — аргумент переменной длины, он может быть опущен. Если `how` присутствует, он может содержать одно или несколько значений, перечисленных в табл. 23.11, которые действительны для всех файловых систем.

Таблица 23.11. Параметры копирования файла, допустимые во всех файловых системах

Значение	Описание
StandardCopyOption.COPY_ATTRIBUTES	Запросить копирование атрибутов файла
LinkOption.NOFOLLOW_LINKS	Не следовать по символическим ссылкам
StandardCopyOption.REPLACE_EXISTING	Перезаписывать существующий файл

В зависимости от реализации могут поддерживаться и другие параметры.

В показанной ниже программе демонстрируется применение метода copy(). Исходный и целевой файлы передаются в командной строке, причем исходный файл указывается первым. Обратите внимание, насколько программа компактна. Возможно, вам захочется сравнить эту версию программы копирования файлов с версией из главы 13. Вы обнаружите, что в показанной здесь версии для NIO часть программы, которая фактически копирует файл, значительно короче.

```
// Копирование файла с использованием системы NIO.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class NIOCopy {
  public static void main(String[] args) {
    if(args.length != 2) {
      System.out.println("Использование: copy исходный-файл целевой-файл");
      return;
    }

    try {
      Path source = Path.of(args[0]);
      Path target = Path.of(args[1]);

      // Копировать файл.
      Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
    } catch(InvalidPathException e) {
      System.out.println("Ошибка в пути: " + e);
    } catch (IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

Использование системы NIO для ввода-вывода, основанного на потоках

Начиная с версии NIO.2, систему NIO можно применять для открытия потока ввода-вывода. При наличии объекта реализации Path необходимо открыть файл, вызвав newInputStream() или newOutputStream(), которые являются статическими методами, определенными в классе Files. Они возвращают поток данных, подключенный к указанному файлу. В любом случае с потоком можно работать так, как было описано в главе 21, с помощью тех же самых приемов. Преимущество использования Path для открытия файла связано с доступностью всех функциональных средств, определенных в NIO.

Чтобы открыть файл для потокового ввода, следует вызвать метод Files.newInputStream():

```
static InputStream newInputStream(Path path, OpenOption ... how)
  throws IOException
```

В path указывается файл, который нужно открыть, а в how — способ его открытия, который определяется одним или несколькими значениями из описанного ранее перечисления StandardOpenOption. (Разумеется, должны применяться только те параметры, которые относятся к потоку ввода.) Если параметры не указаны, тогда файл открывается, как если бы передавалось значение StandardOpenOption.READ.

После открытия файла можно использовать любой метод из числа определенных в InputStream, например, read() для чтения байтов из файла.

В следующей программе демонстрируется применение потокового ввода-вывода на основе NIO. Фактически она представляет собой переделанную программу ShowFile из главы 13 с целью использования функциональных средств NIO для открытия файла и получения потока. Как видите, она очень похожа на первоначальную версию, но только здесь применяется Path и newInputStream().

```
/* Отображение содержимого текстового файла с применением потокового
   ввода-вывода на основе NIO.
   Для использования программы укажите имя файла, который хотите просмотреть.
   Например, чтобы увидеть содержимое файла по имени TEST.TXT,
   введите следующую командную строку:

   java ShowFile TEST.TXT
*/
import java.io.*;
import java.nio.file.*;
class ShowFile {
  public static void main(String[] args)
  {
    int i;
    // Удостовериться, что имя файла было указано.
    if(args.length != 1) {
      System.out.println("Использование: ShowFile имя-файла");
      return;
    }
```

```
    // Открыть файл и получить связанный с ним поток.
    try ( InputStream fin = Files.newInputStream(Path.of(args[0])) )
    {
      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);
    } catch(InvalidPathException e) {
      System.out.println("Ошибка в пути: " + e);
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: "    + e);
    }
  }
}
```

Так как метод `newInputStream()` возвращает обычный поток, его можно использовать подобно любому другому потоку. Скажем, этот поток можно поместить внутрь буферизованного потока, такого как `BufferedInputStream`, и тем самым обеспечить буферизацию:

```
new BufferedInputStream(Files.newInputStream(Path.of(args[0])))
```

Теперь все операции чтения автоматически буферизуются.

Чтобы открыть файл для вывода, необходимо применить метод `Files.newOutputStream()`:

```
static OutputStream newOutputStream(Path path, OpenOption ... how)
  throws IOException
```

В `path` указывается файл, который нужно открыть, а в `how` — способ его открытия, который определяется одним или несколькими значениями из описанного ранее перечисления `StandardOpenOption`. (Конечно, должны использоваться только те параметры, которые относятся к потоку вывода.) Если параметры не указаны, тогда файл открывается, как если бы передавались значения `StandardOpenOption.WRITE`, `StandardOpenOption.CREATE` и `StandardOpenOption.TRUNCATE_EXISTING`.

Методология работы с методом `newOutputStream()` аналогична показанной ранее для метода `newInputStream()`. После открытия файла можно применять любой метод, определенный в `OutputStream`, скажем, `write()` для записи байтов в файл. Кроме того, можно поместить поток данных внутрь `BufferedOutputStream` с целью его буферизации.

Метод `newOutputStream()` в действии демонстрируется в показанной ниже программе, где осуществляется запись алфавита в файл по имени `test.txt`. Обратите внимание на использование буферизованного ввода-вывода.

```
// Демонстрация работы потокового вывода на основе NIO.

import java.io.*;
import java.nio.file.*;

class NIOStreamWrite {
  public static void main(String[] args)
  {
    // Открыть файл и получить связанный с ним поток данных.
```

```
    try ( OutputStream fout =
      new BufferedOutputStream( Files.newOutputStream(Path.of("test.txt"))) )
    {
      // Записать несколько байтов в поток.
      for(int i=0; i < 26; i++)
        fout.write((byte)('A' + i));
    } catch(InvalidPathException e) {
      System.out.println("Ошибка в пути: " + e);
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

Использование системы NIO для операций с путями и файловой системой

В начале главы 22 обсуждался класс `File` из пакета `java.io`. Как там объяснялось, класс `File` имеет дело с файловой системой и разнообразными атрибутами, ассоциированными с файлом, вроде того, является ли файл доступным только для чтения, скрытым и т.д. Он также применялся для получения информации о пути к файлу. Хотя класс `File` по-прежнему вполне приемлем, интерфейсы и классы, определенные в версии NIO.2, предлагают более эффективный способ решения таких задач. Среди их преимуществ — поддержка символических ссылок, улучшенная поддержка обхода дерева каталогов и усовершенствованная обработка метаданных. В последующих разделах приведены примеры двух распространенных операций с файловой системой: получение информации о пути и файле, а также получение содержимого каталога.

Помните! Если вы хотите изменить код, чтобы использовать вместо `java.io.File` интерфейс `Path`, тогда можете с помощью метода `toPath()` получить из экземпляра `File` объект реализации `Path`.

Получение информации о пути и файле

Информацию о пути можно получить с применением методов, определенных в `Path`. Некоторые атрибуты, ассоциированные с файлом, который описан объектом реализации `Path` (например, является ли файл скрытым), получаются посредством методов, определенных в классе `Files`. Здесь используются методы `getName()`, `getParent()` и `toAbsolutePath()` интерфейса `Path`, а также методы `isExecutable()`, `isHidden()`, `isReadable()`, `isWritable()` и `exists()` класса `Files`, которые были кратко описаны в табл. 23.5 и 23.6 ранее в главе.

Внимание! Методы вроде `isExecutable()`, `isReadable()`, `isWritable()` и `exists()` должны применяться осторожно, т.к. после их вызова состояние файловой системы может измениться и тогда программа станет функционировать некорректно. Ситуация подобного рода способна повлечь за собой последствия в плане безопасности.

Другие файловые атрибуты можно получить, запросив список атрибутов вызовом метода `Files.readAttributes()`. В программе метод `Files.readAttributes()` вызывается для получения объекта реализации `BasicFileAttributes`, ассоциированного с файлом, но общий подход применим и к другим типам атрибутов.

В следующей программе демонстрируется использование ряда методов `Path` и `Files`, а также нескольких методов, предоставляемых `BasicFileAttributes`. В программе предполагается, что в каталоге `examples`, который должен быть подкаталогом текущего каталога, существует файл по имени `test.txt`.

```
// Получение информации о пути и файле.
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class PathDemo {
  public static void main(String[] args) {
    Path filepath = Path.of("examples\\test.txt");
    System.out.println("Имя файла: " + filepath.getName(1));
    System.out.println("Путь: " + filepath);
    System.out.println("Абсолютный путь: " + filepath.toAbsolutePath());
    System.out.println("Родительский каталог: " + filepath.getParent());

    if(Files.exists(filepath))
      System.out.println("Файл существует");
    else
      System.out.println("Файл не существует");

    try {
      if(Files.isHidden(filepath))
        System.out.println("Файл является скрытым");
      else
        System.out.println("Файл не является скрытым");
    } catch(IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }

    Files.isWritable(filepath);
    System.out.println("Файл допускает запись");

    Files.isReadable(filepath);
    System.out.println("Файл допускает чтение");

    try {
      BasicFileAttributes attribs =
        Files.readAttributes(filepath, BasicFileAttributes.class);

      if(attribs.isDirectory())
        System.out.println("Файл является каталогом");
      else
        System.out.println("Файл не является каталогом");

      if(attribs.isRegularFile())
        System.out.println("Файл является обычным");
      else
        System.out.println("Файл не является обычным");
```

```
      if(attribs.isSymbolicLink())
        System.out.println("Файл является символической ссылкой");
      else
        System.out.println("Файл не является символической ссылкой");
      System.out.println("Время последней модификации файла: "
        + attribs.lastModifiedTime());
      System.out.println("Размер файла в байтах: " + attribs.size());
    } catch(IOException e) {
      System.out.println("Ошибка при чтении атрибутов: " + e);
    }
  }
}
```

Запустив программу из каталога по имени `MyDir`, в котором есть подкаталог `examples`, содержащий файл `test.txt`, вы увидите вывод, аналогичный показанному ниже. (Конечно, получаемая вами информация будет отличаться.)

```
Имя файла: test.txt
Путь: examples\test.txt
Абсолютный путь: C:\MyDir\examples\test.txt
Родительский каталог: examples
Файл существует
Файл не является скрытым
Файл допускает запись
Файл допускает чтение
Файл не является каталогом
Файл является обычным
Файл не является символической ссылкой
Время последней модификации файла: 2017-01-01T18:20:46.380445Z
Размер файла в байтах: 18
```

В случае работы на компьютере, который поддерживает файловую систему FAT (т.е. файловую систему DOS), можете опробовать методы, определенные в `DosFileAttributes`. При работе на системе, совместимой с POSIX, опробуйте методы из `PosixFileAttributes`.

Получение содержимого каталога

Если путь описывает каталог, тогда прочитать содержимое этого каталога можно с помощью статических методов, определенных в `Files`, для чего сначала получить поток каталога, вызвав метод `newDirectoryStream()` и передав ему объект реализации `Path`, который описывает каталог. Вот одна из форм метода `newDirectoryStream()`:

```
static DirectoryStream<Path> newDirectoryStream(Path dirPath)
  throws IOException
```

В `dirPath` инкапсулирован путь к каталогу. Метод возвращает объект реализации `DirectoryStream<Path>`, который можно применять для получения содержимого каталога. Метод `newDirectoryStream()` генерирует исключение `IOException`, если возникла ошибка ввода-вывода, и `NotDirectoryException` (подкласс `IOException`), если указанный путь не является каталогом. Возможна также генерация исключения `SecurityException`, когда доступ к каталогу запрещен.

Интерфейс DirectoryStream<Path> расширяет AutoCloseable, а потому им можно управлять с помощью оператора try с ресурсами. Вдобавок он расширяет Iterable<Path>, т.е. содержимое каталога можно получить, проходя по DirectoryStream. При проходе каждая запись каталога представлена объектом реализации Path. Простой способ выполнить проход по DirectoryStream предусматривает использование цикла for в стиле "for-each". Однако важно понимать, что итератор, реализованный посредством DirectoryStream<Path>, можно получить только раз для каждого экземпляра. Таким образом, метод iterator() допускается вызывать лишь однократно, а цикл for в стиле "for-each" может быть выполнен только однократно.

В следующей программе отображается содержимое каталога по имени MyDir:

```
// Отображение содержимого каталога.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class DirList {
  public static void main(String[] args) {
    String dirname = "\\MyDir";

    // Получить и управлять потоком каталога внутри блока try.
    try ( DirectoryStream<Path> dirstrm =
          Files.newDirectoryStream(Path.of(dirname)) )
    {
      System.out.println("Каталог " + dirname);

      // Поскольку DirectoryStream расширяет Iterable,
      // для отображения содержимого
      // каталога можно использовать цикл for в стиле "for-each".
      for(Path entry : dirstrm) {
        BasicFileAttributes attribs =
          Files.readAttributes(entry, BasicFileAttributes.class);

        if(attribs.isDirectory())
          System.out.print("<КАТ> ");
        else
          System.out.print("     ");

        System.out.println(entry.getName(1));
      }
    } catch(InvalidPathException e) {
      System.out.println("Ошибка в пути: " + e);
    } catch(NotDirectoryException e) {
      System.out.println(dirname + " не является каталогом.");
    } catch (IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

Ниже показан пример вывода, сгенерированный программой:

```
Каталог \MyDir
     DirList.class
     DirList.java
<КАТ> examples
     Test.txt
```

Фильтровать содержимое каталога можно двумя способами. Проще всего применить такую версию newDirectoryStream():

```
static DirectoryStream<Path>
  newDirectoryStream(Path dirPath, String wildcard) throws IOException
```

С помощью данной версии будут получены только файлы, которые соответствуют шаблону имен файлов, указанному в wildcard. В wildcard можно указать либо полное имя файла, либо *шаблон поиска* — строку с общим шаблоном, использующим знакомые символы подстановки * и ?, которые будут давать совпадение с одним или несколькими файлам. Символы подстановки * и ? соответствуют нулю или большему числу любых символов и любому одному символу. В шаблоне поиска также распознаются элементы, описанные в табл. 23.12.

Таблица 23.12. Элементы, распознаваемые в шаблоне поиска

Элемент	Описание
**	Соответствует нулю или большему числу любых символов в каталогах
[*символы*]	Соответствует любому символу в элементе *символы*. Символы * и ? будут трактоваться как обычные символы, а не символы подстановки. С использованием дефиса можно указывать диапазон, например, [x-z]
{*список-шаблонов*}	Соответствует любому шаблону, указанному в элементе *список-шаблонов*

Указать символ * или ? можно с применением * или \?. Для указания \ используется \\. Полезно поэкспериментировать с шаблоном поиска, заменив вызов newDirectoryStream() в предыдущей программе следующим образом:

```
Files.newDirectoryStream(Path.of(dirname), "{Path,Dir}*.{java,class}")
```

Такой вызов получает поток каталога, содержащий только файлы, имена которых начинаются с Path или Dir и используют расширение .java или .class. Таким образом, произойдет совпадение с такими именами, как DirList.java и PathDemo.java, но не MyPathDemo.java.

Еще один способ фильтрации каталога предусматривает применение показанной ниже версии newDirectoryStream():

```
static DirectoryStream<Path> newDirectoryStream(Path dirPath,
  DirectoryStream.Filter<? super Path> filefilter)
    throws IOException
```

Здесь `DirectoryStream.Filter` представляет собой интерфейс, в котором определен следующий метод:

```
boolean accept(T entry) throws IOException
```

В данном случае типом `T` будет `Path`. Если необходимо включить `entry` в список, тогда понадобится возвратить `true`, а иначе `false`. Преимущество такой формы `newDirectoryStream()` связано с возможностью фильтрации каталога на основе чего-то другого помимо имени файла. Скажем, каталог можно фильтровать по размеру, дате создания, дате модификации или атрибуту и т.д. Процесс демонстрируется в приведенной далее программе, где выводятся только файлы, допускающие запись:

```
// Отображение только тех файлов из каталога, которые допускают запись.
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
class DirList {
  public static void main(String[] args) {
    String dirname = "\\MyDir";
    // Создать фильтр, который возвращает true только для файлов,
    // допускающих запись.
    DirectoryStream.Filter<Path> how = new DirectoryStream.Filter<Path>() {
      public boolean accept(Path filename) throws IOException {
        if(Files.isWritable(filename)) return true;
        return false;
      }
    };
    // Получить и управлять потоком каталога для файлов, допускающих запись.
    try (DirectoryStream<Path> dirstrm =
        Files.newDirectoryStream(Path.of(dirname), how) )
    {
      System.out.println("Каталог " + dirname);
      for(Path entry : dirstrm) {
        BasicFileAttributes attribs =
          Files.readAttributes(entry, BasicFileAttributes.class);
        if(attribs.isDirectory())
          System.out.print("<КАТ> ");
        else
          System.out.print("     ");
        System.out.println(entry.getName(1));
      }
    } catch(InvalidPathException e) {
      System.out.println("Ошибка в пути: " + e);
    } catch(NotDirectoryException e) {
      System.out.println(dirname + " не является каталогом.");
    } catch (IOException e) {
      System.out.println("Ошибка ввода-вывода: " + e);
    }
  }
}
```

Использование метода walkFileTree() для вывода дерева каталогов

В предыдущих примерах получалось содержимое одиночного каталога. Тем не менее, иногда требуется получить список файлов в дереве каталогов. Раньше это было не особо легко, но NIO.2 упрощает задачу, поскольку теперь для обработки дерева каталогов можно применять метод `walkFileTree()`, определенный в `Files`. Он имеет две формы. Ниже показана та, что используется в настоящей главе:

```
static Path walkFileTree(Path root, FileVisitor<? super Path> fv)
  throws IOException
```

В `root` указывается путь к начальной точке обхода дерева каталогов. В `fv` передается экземпляр реализации `FileVisitor`. Реализация `FileVisitor` определяет способ обхода дерева каталогов и предоставляет доступ к информации о каталогах. В случае возникновения ошибки ввода-вывода генерируется исключение `IOException`. Также может быть сгенерировано исключение `SecurityException`.

`FileVisitor` — это интерфейс, который определяет, каким образом файлы посещаются при обходе дерева каталогов. Он является обобщенным интерфейсом со следующим объявлением:

```
interface FileVisitor<T>
```

Для использования в `walkFileTree()` типом `T` будет `Path` (или любой тип, производный от `Path`). Методы, определенные в интерфейсе `FileVisitor`, кратко описаны в табл. 23.13.

Таблица 23.13. Методы, определенные в интерфейсе `FileVisitor`

Метод	Описание
`FileVisitResult postVisitDirectory (T dir, IOException exc) throws IOException`	Вызывается после посещения каталога. Каталог передается в `dir`, а любое исключение `IOException` — в `exc`. Если `exc` равно `null`, то исключения не возникали. Возвращает результат типа `FileVisitResult`
`FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attribs) throws IOException`	Вызывается перед посещением каталога. Каталог передается в `dir`, а ассоциированные с каталогом атрибуты — в `attribs`. Возвращает результат типа `FileVisitResult`. Для исследования каталога необходимо возвратить `FileVisitResult.CONTINUE`

Окончание табл. 23.13

Метод	Описание
`FileVisitResult visitFile (T file, BasicFileAttributes attribs) throws IOException`	Вызывается при посещении файла. Файл передается в `file`, а ассоциированные с файлом атрибуты — в `attribs`. Возвращает результат типа `FileVisitResult`
`FileVisitResult visitFileFailed (T file, IOException exc) throws IOException`	Вызывается при неудачной попытке посещения файла. Файл, который не удалось посетить, передается в `file`, а любое исключение `IOException` — в `exc`. Возвращает результат типа `FileVisitResult`

Обратите внимание, что каждый метод возвращает результат типа `FileVisitResult`. В перечислении `FileVisitResult` определены следующие значения:

```
CONTINUE        SKIP_SIBLINGS        SKIP_SUBTREE        TERMINATE
```

Для продолжения обхода каталога и подкаталогов любой метод интерфейса `FileVisitor` должен возвращать значение `CONTINUE`. Что касается метода `preVisitDirectory()`, то для пропуска каталога и его одноуровневых элементов, а также предотвращения вызова `postVisitDirectory()` необходимо возвратить `SKIP_SIBLINGS`. Чтобы пропустить только каталог и подкаталоги, понадобится возвратить `SKIP_SUBTREE`. Чтобы остановить обход каталога, нужно возвратить `TERMINATE`.

Хотя определенно можно создать собственный класс для посещения файлов, который реализует методы, определенные в `FileVisitor`, обычно поступать так не имеет смысла, поскольку класс `SimpleFileVisitor` предлагает простую реализацию. Можно всего лишь переопределить стандартную реализацию необходимого метода или методов. Ниже приведен краткий пример, иллюстрирующий такой процесс. В нем отображаются все файлы в дереве каталогов, корнем которого является `\MyDir`. Обратите внимание, насколько компактной получилась программа.

```
// Простой пример, в котором для отображения дерева каталогов
// используется метод walkFileTree().

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

// Создать специальную версию класса SimpleFileVisitor,
// в которой переопределяется метод visitFile().
```

```
class MyFileVisitor extends SimpleFileVisitor<Path> {
  public FileVisitResult visitFile(Path path, BasicFileAttributes attribs)
    throws IOException
  {
    System.out.println(path);
    return FileVisitResult.CONTINUE;
  }
}

class DirTreeList {
  public static void main(String[] args) {
    String dirname = "\\MyDir";
    System.out.println("Дерево каталогов, начиная с " + dirname + ":\n");
    try {
      Files.walkFileTree(Path.of(dirname), new MyFileVisitor());
    } catch (IOException exc) {
      System.out.println("Ошибка ввода-вывода");
    }
  }
}
```

Вот пример вывода, который выдает программа при запуске в том же каталоге MyDir. В этом примере подкаталог по имени examples содержит один файл MyProgram.java.

```
Дерево каталогов, начиная с \MyDir:

\MyDir\DirList.class
\MyDir\DirList.java
\MyDir\examples\MyProgram.java
\MyDir\Test.txt
```

Класс MyFileVisitor в программе расширяет SimpleFileVisitor, переопределяя только метод visitFile(), который здесь просто отображает файлы, но реализовать более сложную функциональность очень легко. Скажем, можно фильтровать файлы или выполнять с ними действия наподобие копирования их на резервное устройство. Ради ясности именованный класс применялся для переопределения метода visitFile(), но допускается также использовать анонимный внутренний класс.

И последнее замечание: отслеживать изменения в каталоге можно с помощью интерфейса java.nio.file.WatchService.

ГЛАВА

24 Работа в сети

Язык Java ассоциировался с программированием для Интернета с самого начала своего существования. На то имеется ряд причин, не последней из которых является его способность генерировать безопасный, межплатформенный и переносимый код. Однако одной из важнейших причин того, что Java стал основным языком сетевого программирования, являются классы из пакета `java.net`, предоставляющие удобные средства, с помощью которых программисты всех уровней квалификации могут получать доступ к сетевым ресурсам. Начиная с версии JDK 11, язык Java также обеспечивает расширенную сетевую поддержку HTTP-клиентов через пакет `java.net.http` в одноименном модуле. Инфраструктура называется HTTP Client API и еще больше расширяет сетевые возможности Java.

В настоящей главе исследуется пакет `java.net`, а в конце главы представлен пакет `java.http.net`. Важно отметить, что работа в сети — обширная и местами сложная тема. В этой книге невозможно обсудить все возможности, содержащиеся в упомянутых двух пакетах. Взамен основное внимание в главе уделяется нескольким основным классам и интерфейсам из указанных пакетов.

Основы работы в сети

Перед тем как начать, полезно рассмотреть некоторые ключевые понятия и термины, связанные с сетями. В основе сетевой поддержки Java лежит концепция *сокета*. Сокет идентифицирует конечную точку в сети. Парадигма сокетов была частью выпуска операционной системы UNIX версии 4.2BSD Berkeley в начале 1980-х годов, из-за чего также применяется термин *сокет Беркли*. Сокеты лежат в основе современной работы в сети, т.к. сокет позволяет одному компьютеру одновременно обслуживать множество разных клиентов и поддерживать различные виды информации. Цель достигается за счет использования *порта*, который является нумерованным сокетом на отдельной машине. Говорят, что серверный процесс "прослушивает" порт до тех пор, пока к нему не подключится клиент. Серверу разрешено принимать запросы от множества клиентов, подключенных к одному и тому же номеру порта, хотя каждый сеанс

уникален. Для управления несколькими клиентскими подключениями серверный процесс должен быть многопоточным либо располагать другими средствами мультиплексирования одновременных операций ввода-вывода.

Связь через сокеты осуществляется согласно некоторому протоколу. *Протокол Интернета* (Internet Protocol — IP) представляет собой низкоуровневый протокол маршрутизации, который разбивает данные на небольшие пакеты и отправляет их по определенному адресу через сеть, не гарантируя доставку указанных пакетов в пункт назначения. *Протокол управления передачей* (Transmission Control Protocol — TCP) является протоколом более высокого уровня, который надежно объединяет эти пакеты, сортируя и повторно посылая их по мере необходимости для надежной передачи данных. Третий протокол, *протокол пользовательских дейтаграмм* (User Datagram Protocol — UDP), находится рядом с TCP и может применяться напрямую для поддержки быстрой и ненадежной передачи пакетов без установления подключения.

После того как подключение установлено, в действие вступает протокол более высокого уровня, который зависит от того, какой порт используется. Протокол TCP/IP резервирует младшие 1024 порта для специальных протоколов. Некоторые из них могут быть вам знакомы. Например, порт номер 21 предназначен для FTP, 23 — для Telnet, 25 — для электронной почты, 43 — для whois, 80 — для HTTP и 119 — для сетевых новостей. Каждый протокол самостоятельно определяет, каким образом клиент должен взаимодействовать с портом.

Скажем, HTTP представляет собой протокол, который веб-браузеры и серверы применяют для передачи гипертекстовых страниц и изображений. Он является довольно простым протоколом для веб-сервера с базовым просмотром страниц. Давайте рассмотрим, как работает протокол HTTP. Когда клиент запрашивает файл из HTTP-сервера (действие, известное как *посещение*), он просто отправляет имя файла в специальном формате на заранее определенный порт и читает содержимое файла. Сервер также отвечает кодом состояния, чтобы сообщить клиенту, может ли запрос быть удовлетворен, а если нет, то по какой причине.

Ключевым компонентом Интернета считается *адрес*. Его имеет каждый компьютер в Интернете. Адрес Интернета — это число, которое однозначно идентифицирует каждый компьютер во всемирной сети. Первоначально все адреса Интернета состояли из 32-битных значений, организованных в виде четырех 8-битных групп. Такой тип адреса был определен в IPv4 (протокол Интернета версии 4). Тем не менее, в игру вступила более новая схема адресации, называемая IPv6 (протокол Интернета версии 6). Для представления адреса протокол IPv6 использует 128-битное значение, организованное в восемь 16-битных фрагментов. Хотя появление протокола IPv6 связано с несколькими причинами и он обладает рядом преимуществ, главный момент заключается в том, что IPv6 поддерживает гораздо большее адресное пространство по сравнению с IPv4. К счастью, при написании кода на Java обычно не нужно беспокоиться о том, какие адреса применяются — IPv4 или IPv6, поскольку все детали обрабатываются автоматически.

В точности как числа IP-адреса описывают сетевую иерархию, имя адреса Интернета, называемое его *доменным именем*, описывает местоположение машины в пространстве имен. Например, имя `www.HerbSchildt.com` находится в домене верхнего уровня `com` (используется коммерческими сайтами в США); оно называется `HerbSchildt`, а `www` идентифицирует сервер для вебзапросов. Доменное имя Интернета сопоставляется с IP-адресом *службой доменных имен* (Domain Naming Service — DNS), что позволяет пользователям работать с доменными именами, но Интернет оперирует IP-адресами.

Классы и интерфейсы пакета `java.net` для работы в сети

Пакет `java.net` содержит исходные функциональные средства для работы в сети, которые доступны, начиная с версии Java 1.0. Он обеспечивает поддержку протокола TCP/IP как за счет расширения имеющегося интерфейса потокового ввода-вывода, представленного в главе 22, так и путем добавления средств, необходимых для построения объектов ввода-вывода через сеть. В Java поддерживаются семейства протоколов TCP и UDP. Протокол TCP применяется для надежного потокового ввода-вывода по сети. Протокол UDP поддерживает более простую и, следовательно, более быструю модель, ориентированную на дейтаграммы типа "точка-точка". Ниже перечислены классы, которые определены в пакете `java.net`:

`Authenticator`	`InetAddress`	`SocketAddress`
`CacheRequest`	`InetSocketAddress`	`SocketImpl`
`CacheResponse`	`InterfaceAddress`	`SocketPermission`
`ContentHandler`	`JarURLConnection`	`StandardSocketOption`
`CookieHandler`	`MulticastSocket`	`UnixDomainSocketAddress`
`CookieManager`	`NetPermission`	`URI`
`DatagramPacket`	`NetworkInterface`	`URL`
`DatagramSocket`	`PasswordAuthentication`	`URLClassLoader`
`DatagramSocketImpl`	`Proxy`	`URLConnection`
`HttpCookie`	`ProxySelector`	`URLDecoder`
`HttpURLConnection`	`ResponseCache`	`URLEncoder`
`IDN`	`SecureCacheResponse`	`URLPermission`
`Inet4Address`	`ServerSocket`	`URLStreamHandler`
`Inet6Address`	`Socket`	

А вот интерфейсы из пакета `java.net`:

`ContentHandlerFactory`	`FileNameMap`	`SocketOptions`
`CookiePolicy`	`ProtocolFamily`	`URLStreamHandlerFactory`
`CookieStore`	`SocketImplFactory`	
`DatagramSocketImplFactory`	`SocketOption`	

Начиная с версии JDK 9, пакет `java.net` входит в состав модуля `java.base`. В последующих разделах рассматриваются основные сетевые классы и предлагается несколько соответствующих примеров. Освоив основные сетевые классы, вы сможете легко изучить остальное самостоятельно.

InetAddress

Класс `InetAddress` используется для инкапсуляции числового IP-адреса и доменного имени для данного адреса. Для взаимодействия с классом `InetAddress` применяется имя IP-хоста, что более удобно и понятно, нежели его IP-адрес. Класс `InetAddress` скрывает число внутри и способен обрабатывать адреса IPv4 и IPv6.

Фабричные методы

Класс `InetAddress` не имеет конструкторов. Для создания объекта `InetAddress` необходимо использовать один из доступных фабричных методов. Как объяснялось ранее в книге, фабричные методы представляют собой просто соглашение, в соответствии с которым статические методы в классе возвращают экземпляр этого класса. Они позволяют избавиться от перегрузки конструктора с различными списками параметров и за счет уникальных имен методов делают результаты более ясными. Ниже показаны три часто применяемых фабричных метода `InetAddress`:

```
static InetAddress getLocalHost()
  throws UnknownHostException
static InetAddress getByName(String hostName)
  throws UnknownHostException
static InetAddress[] getAllByName(String hostName)
  throws UnknownHostException
```

Метод `getLocalHost()` просто возвращает объект `InetAddress`, который представляет локальный хост. Метод `getByName()` возвращает объект `InetAddress` для переданного ему имени хоста. Если перечисленные выше методы не могут распознать имя узла, тогда они генерируют исключение `UnknownHostException`.

Обычно в Интернете одиночное имя используется для представления нескольких машин. В мире веб-серверов это один из способов обеспечить некоторую степень масштабирования. Фабричный метод `getAllByName()` возвращает массив объектов `InetAddress`, представляющий все адреса, в которые преобразуется конкретное имя. Он также генерирует исключение `UnknownHostException`, если имя не удалось преобразовать хотя бы в один адрес.

В класс `InetAddress` также определен фабричный метод `getByAddress()`, который принимает IP-адрес и возвращает объект `InetAddress`. Допускается указывать адрес IPv4 или IPv6.

В следующем примере выводятся адреса и имена локальной машины и двух веб-сайтов в Интернете:

```
// Демонстрация использования InetAddress.
import java.net.*;
class InetAddressTest
{
  public static void main(String[] args) throws UnknownHostException {
    InetAddress Address = InetAddress.getLocalHost();
    System.out.println(Address);
    Address = InetAddress.getByName("www.HerbSchildt.com");
    System.out.println(Address);
    InetAddress[] SW = InetAddress.getAllByName("www.nba.com");
    for (int i=0; i<SW.length; i++)
      System.out.println(SW[i]);
  }
}
```

Вот результат работы программы. (Разумеется, полученный вами вывод может немного отличаться.)

```
default/166.203.115.212
www.HerbSchildt.com/216.92.65.4
www.nba.com/23.67.86.30
www.nba.com/2600:1407:2800:3a4:0:0:0:1f51
www.nba.com/2600:1407:2800:3ad:0:0:0:1f51
```

Методы экземпляра

В классе `InetAddress` определено несколько других методов, которые можно применять для объектов, возвращаемых только что рассмотренными методами. Избранные методы кратко описаны в табл. 24.1.

Таблица 24.1. Избранные методы, определенные в классе `InetAddress`

Метод	Описание
`boolean equals (Object other)`	Возвращает `true`, если этот объект имеет такой же адрес Интернета, как и другой
`byte[] getAddress()`	Возвращает байтовый массив, который представляет IP-адрес объекта в сетевом порядке следования байтов
`String getHostAddress()`	Возвращает строку, которая представляет адрес хоста, ассоциированного с объектом `InetAddress`
`String getHostName()`	Возвращает строку, которая представляет имя хоста, ассоциированного с объектом `InetAddress`
`boolean isMulticastAddress()`	Возвращает `true`, если этот адрес является групповым, или `false` в противном случае
`String toString()`	Возвращает строку, в которой удобно представлены имя и IP-адрес хоста

Адреса Интернета ищутся в группе иерархических серверов с кешированием, т.е. локальному компьютеру может быть автоматически известно конкретное сопоставление имени с IP-адресом, скажем, для себя и соседних серверов. Для других имен он может запрашивать у локального DNS-сервера информацию об IP-адресах. Если у данного сервера нет определенного адреса, тогда локальный компьютер может перейти на удаленный сайт и запросить его. Процесс может продолжаться вплоть до корневого сервера и занимать длительное время, поэтому разумно структурировать свой код так, чтобы информацию об IP-адресе кешировать локально, а не искать ее повторно.

Inet4Address и Inet6Address

Язык Java включает поддержку адресов IPv4 и IPv6, из-за чего были созданы два подкласса `InetAddress`: `Inet4Address` и `Inet6Address`. Класс `Inet4Address` представляет традиционный адрес IPv4. Класс `Inet6Address` инкапсулирует новый адрес IPv6. Поскольку они являются подклассами `InetAddress`, ссылка типа `InetAddress` может ссылаться на любой из них. Это один из способов, с помощью которого в Java удалось добавить функциональность IPv6, не нарушив работу существующего кода и не определив множество дополнительных классов. По большей части при работе с IP-адресами можно просто использовать класс `InetAddress`, т.к. он поддерживает оба стиля.

Клиентские сокеты TCP/IP

Сокеты TCP/IP применяются для реализации надежных, двунаправленных, постоянных, двухточечных, потоковых соединений между хостами в Интернете. Сокет можно использовать для подключения системы ввода-вывода Java к другим программам, которые могут находиться либо на локальном компьютере, либо на любом другом компьютере в Интернете с учетом ограничений безопасности.

В Java существуют два типа сокетов TCP, которые ориентированы на серверы и на клиенты. Класс `ServerSocket` спроектирован как "прослушиватель", ожидающий подключения клиентов, прежде чем что-либо делать. Таким образом, класс `ServerSocket` предназначен для серверов, а `Socket` — для клиентов. Класс `Socket` спроектирован так, чтобы подключаться к серверным сокетам и инициировать обмен данными. Поскольку клиентские сокеты чаще всего применяются в приложениях Java, именно они рассматриваются в главе.

Создание объекта `Socket` неявно устанавливает подключение между клиентом и сервером. Методы или конструкторы, которые бы явно раскрыли детали установления такого подключения, отсутствуют. В табл. 24.2 кратко описаны два конструктора, используемые для создания клиентских сокетов.

Таблица 24.2. Конструкторы, определенные в классе `Socket`

Метод	Описание
`Socket(String hostName, int port) throws UnknownHostException, IOException`	Создает сокет, подключенный к указанному хосту и порту
`Socket(InetAddress ipAddress, int port) throws IOException`	Создает сокет с применением существующего объекта `InetAddress` и порта

В классе `Socket` определено несколько методов экземпляра. Например, объект `Socket` можно в любой момент проверить на наличие связанной с ним информации об адресе и порте посредством методов из табл. 24.3.

Таблица 24.3. Методы, определенные в классе `Socket`

Метод	Описание
`InetAddress getInetAddress()`	Возвращает объект `InetAddress`, ассоциированный с объектом `Socket`, или `null`, если сокет не подключен
`int getPort()`	Возвращает удаленный порт, к которому подключен вызывающий объект `Socket`, или 0, если сокет не подключен
`int getLocalPort()`	Возвращает локальный порт, к которому привязан вызывающий объект `Socket`, или –1, если сокет не привязан

Получить доступ к потокам ввода и вывода, связанным с сокетом, можно с применением методов `getInputStream()` и `getOuputStream()`, которые кратко описаны в табл. 24.4. Оба метода генерируют исключение `IOException`, если сокет был объявлен недействительным из-за утраты подключения. Такие потоки используются точно так же, как потоки ввода-вывода, описанные в главе 22, для отправки и получения данных.

Таблица 24.4. Методы `getInputStream()` и `getOuputStream()`

Метод	Описание
`InputStream getInputStream() throws IOException`	Возвращает объект `InputStream`, ассоциированный с вызывающим объектом
`OutputStream getOutputStream() throws IOException`	Возвращает объект `OutputStream`, ассоциированный с вызывающим объектом

Доступно несколько других методов, в том числе connect(), который позволяет указать новое подключение, isConnected(), возвращающий true, если сокет подключен к серверу, isBound(), который возвращает true, если сокет привязан к адресу, и isClosed(), возвращающий true, если сокет закрыт. Метод close() позволяет закрыть сокет. Закрытие сокета приводит к закрытию также потоков ввода-вывода, ассоциированных с сокетом. Кроме того, класс Socket реализует интерфейс AutoCloseable, т.е. сокетом можно управлять с применением оператора try с ресурсами.

В следующей программе демонстрируется простой пример использования класса Socket. Код в ней открывает подключение к порту whois (с номером 43) на сервере InterNIC, отправляет аргумент командной строки через сокет и отображает возвращенные данные. Сервер InterNIC попытается найти зарегистрированное имя домена в Интернете, соответствующее переданному аргументу, после чего отправит обратно IP-адрес и контактную информацию для этого сайта.

```
// Демонстрация использования класса Socket.

import java.net.*;
import java.io.*;

class Whois {
  public static void main(String[] args) throws Exception {
    int c;

    // Создать сокет, подключенный к internic.net, порт 43.
    Socket s = new Socket("whois.internic.net", 43);

    // Получить потоки ввода и вывода.
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    // Сконструировать строку запроса.
    String str = (args.length == 0 ? "MHProfessional.com" : args[0]) + "\n";

    // Преобразовать в байты.
    byte[] buf = str.getBytes();

    // Отправить запрос.
    out.write(buf);

    // Прочитать и отобразить ответ.
    while ((c = in.read()) != -1) {
      System.out.print((char) c);
    }
    s.close();
  }
}
```

Давайте обсудим работу программы. Первым делом создается объект Socket с указанием имени хоста "whois.internic.net" и номера порта 43. Здесь internic.net — веб-сайт InterNIC, обрабатывающий запросы whois, а порт 43 — порт whois. Затем на сокете открываются потоки ввода и вывода. Далее конструируется строка, содержащая имя веб-сайта, информацию о котором нужно получить. Если в командной строке веб-сайт не указан, тогда

применяется "MHProfessional.com". Строка преобразуется в байтовый массив и отправляется из сокета. Ответ читается из сокета и полученные результаты отображаются. Наконец, сокет закрывается, что приводит к закрытию и потоков ввода-вывода.

В рассмотренном примере сокет закрывался вручную вызовом метода `close()`. При наличии современной версии Java можно использовать блок `try` с ресурсами для автоматического закрытия сокета. Например, вот еще один способ реализации метода `main()` из предыдущей программы:

```
// Закрытие сокета посредством оператора try с ресурсами.
public static void main(String[] args) throws Exception {
  int c;

  // Создать сокет, подключенный к internic.net, порт 43.
  // Управлять этим сокетом с помощью блока try с ресурсами.
  try ( Socket s = new Socket("whois.internic.net", 43) ) {

    // Получить потоки ввода и вывода.
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    // Сконструировать строку запроса.
    String str = (args.length == 0 ? "MHProfessional.com" : args[0]) + "\n";
    // Преобразовать в байты.
    byte[] buf = str.getBytes();

    // Отправить запрос.
    out.write(buf);

    // Прочитать и отобразить ответ.
    while ((c = in.read()) != -1) {
      System.out.print((char) c);
    }
  }
  // Сокет теперь закрыт.
}
```

В данной версии сокет автоматически закрывается по окончании блока `try`.

Приведенные в книге примеры должны работать с более ранними версиями Java и потому в них явно вызывается метод `close()`, чтобы четко проиллюстрировать, когда сетевой ресурс может быть закрыт. Однако при написании собственного кода непременно обдумайте возможность применения автоматического управления ресурсами, т.к. оно обеспечивает более рациональный подход. И еще один момент: в предыдущей программе генерируемые в методе `main()` исключения можно было бы обработать, добавив в конец блока `try` с ресурсами соответствующие конструкции `catch`.

На заметку! Ради простоты в примерах, рассматриваемых в этой главе, все генерируемые внутри `main()` исключения просто игнорируются, что позволяет четко проиллюстрировать логику сетевого кода. Тем не менее, в реальном коде исключения должны надлежащим образом обрабатываться.

URL

Предыдущий пример был не особо ясным, потому что в современном Интернете не используются устаревшие протоколы вроде whois, finger и FTP. Речь идет о WWW — "всемирной паутине", или веб-сети. Веб-сеть представляет собой несвязанную совокупность протоколов и файловых форматов более высокого уровня, объединенных в веб-браузере. Один из наиболее важных аспектов веб-сети заключается в том, что Тим Бернерс-Ли изобрел масштабируемый способ обнаружения всех сетевых ресурсов. При условии надежного именования всего в веб-сети данный аспект становится очень мощной парадигмой. Именно для этого предназначен *унифицированный указатель ресурса* (Uniform Resource Locator — URL).

Указатель URL обеспечивает достаточно понятную форму для уникальной идентификации или адресации информации в Интернете. Указатели URL вездесущи; каждый браузер применяет их для идентификации информации в веб-сети. В библиотеке сетевых классов Java класс `URL` предлагает простой и лаконичный API-интерфейс, предназначенный для доступа к информации через Интернет с использованием URL.

Все URL имеют один и тот же базовый формат, хотя допускаются некоторые вариации. Вот два примера: `http://www.HerbSchildt.com/` и `http://www.HerbSchildt.com:80/index.htm`. Спецификация URL основана на четырех компонентах. Первый компонент — применяемый протокол, который отделяется от остальной части URL двоеточием (`:`). Распространенными протоколами являются HTTP, FTP и файловый, хотя в наши дни почти все делается через HTTP (вообще говоря, большинство браузеров будет работать правильно, если в спецификации URL опущена часть `http://`).

Второй компонент — имя хоста или IP-адрес используемого хоста; он отделяется слева двойной косой чертой (`//`) и справа косой чертой (`/`) или дополнительно двоеточием (`:`). Третий компонент — номер порта — является необязательным; он отделяется слева от имени хоста двоеточием (`:`) и справа косой чертой (`/`). (По умолчанию используется предопределенный в протоколе HTTP порт 80, так что конструкция `:80` избыточна.) Четвертый компонент — фактический путь к файлу. Большинство HTTP-серверов добавляют файл по имени `index.html` или `index.htm` к URL, которые относятся непосредственно к ресурсу каталога. Таким образом, URL вида `http://www.HerbSchildt.com/` будет таким же, как `http://www.HerbSchildt.com/index.htm`.

Класс `URL` имеет несколько конструкторов, каждый из которых может генерировать исключение `MalformedURLException`. В одной часто применяемой форме URL указывается с помощью строки, идентичной той, что отображается в браузере:

```
URL(String urlSpecifier) throws MalformedURLException
```

Следующие две формы конструктора позволяют разбивать URL на компоненты:

```
URL(String protocolName, String hostName, int port, String path)
  throws MalformedURLException
URL(String protocolName, String hostName, String path)
  throws MalformedURLException
```

Еще одна часто используемая форма конструктора дает возможность применять существующий URL в качестве ссылочного контекста и затем создать из этого контекста новый URL. На самом деле все довольно просто и удобно:

```
URL(URL urlObj, String urlSpecifier) throws MalformedURLException
```

В показанном далее примере создается URL для страницы веб-сайта HerbSchildt.com, после чего исследуются его свойства:

```
// Демонстрация использования класса URL.

import java.net.*;

class URLDemo {
  public static void main(String[] args) throws MalformedURLException {
    URL hp = new URL(http://www.HerbSchildt.com/WhatsNew");

    System.out.println("Протокол: " + hp.getProtocol());
    System.out.println("Порт: " + hp.getPort());
    System.out.println("Хост: " + hp.getHost());
    System.out.println("Файл: " + hp.getFile());
    System.out.println("Внешняя форма:" + hp.toExternalForm());
  }
}
```

Вот вывод, получаемый из программы:

```
Протокол: http
Порт: -1
Хост: www.HerbSchildt.com
Файл: /WhatsNew
Внешняя форма: http://www.HerbSchildt.com/WhatsNew
```

Обратите внимание на что, для порта выводится -1, т.к. порт не был задан явно. Имея объект URL, можно получить ассоциированные с ним данные. Чтобы получить доступ к фактическому содержимому объекта URL, понадобится создать из него объект URLConnection, используя метод openConnection() класса URL:

```
urlc = url.openConnection()
```

Метод openConnection() имеет следующую общую форму:

```
URLConnection openConnection() throws IOException
```

Он возвращает объект URLConnection, ассоциированный с вызывающим объектом URL. Имейте в виду, что метод openConnection() может сгенерировать исключение IOException.

URLConnection

URLConnection — это универсальный класс, предназначенный для доступа к атрибутам удаленного ресурса. После подключения к удаленному сер-

веру класс URLConnection можно применять для инспектирования свойств удаленного объекта, прежде чем фактически передавать его локально. Такие атрибуты предоставляются спецификацией протокола HTTP и потому имеют смысл только для объектов URL, которые используют протокол HTTP.

В классе URLConnection определено несколько методов, часть которых кратко описана в табл. 24.5.

Таблица 24.5. Избранные методы, определенные в классе URLConnection

Метод	Описание
`int getContentLength()`	Возвращает размер в байтах содержимого, ассоциированного с ресурсом. Если длина недоступна, тогда возвращает -1
`long getContentLengthLong()`	Возвращает размер в байтах содержимого, ассоциированного с ресурсом. Если длина недоступна, тогда возвращает -1
`String getContentType()`	Возвращает тип содержимого, обнаруженного в ресурсе. Это значение поля заголовка content-type. Возвращает null, если тип содержимого недоступен
`long getDate()`	Возвращает дату и время ответа в миллисекундах, прошедших с 1 января 1970 года (GMT)
`long getExpiration()`	Возвращает дату и время срока действия ресурса в миллисекундах, прошедших с 1 января 1970 года (GMT). Если дата срока действия недоступна, тогда возвращает 0
`String getHeaderField (int idx)`	Возвращает значение поля заголовка по индексу idx. (Поля заголовка индексируются, начиная с нуля.) Возвращает null, если значение idx превышает количество полей
`String getHeaderField (String fieldName)`	Возвращает значение поля заголовка с именем fieldName. Возвращает null, если поле заголовка с указанным именем не найдено
`String getHeaderFieldKey(int idx)`	Возвращает ключ поля заголовка по индексу idx. (Поля заголовка индексируются, начиная с нуля.) Возвращает null, если значение idx превышает количество полей
`Map<String, List<String>> getHeaderFields()`	Возвращает карту, содержащую все поля и значения заголовка

Окончание табл. 24.5

Метод	Описание
`long getLastModified()`	Возвращает дату и время последней модификации ресурса в миллисекундах, прошедших с 1 января 1970 года (GMT). Если дата последней модификации недоступна, тогда возвращает 0
`InputStream getInputStream() throws IOException`	Возвращает объект `InputStream`, связанный с ресурсом. Поток `InputStream` можно использовать для получения содержимого ресурса

Обратите внимание, что в классе `URLConnection` определено несколько методов, которые обрабатывают информацию заголовка. Заголовок состоит из пар ключей и значений, представленных в виде строк. С применением метода `getHeaderField()` можно получить значение, ассоциированное с ключом заголовка. С использованием метода `getHeaderFields()` можно получить карту, содержащую все поля заголовка. Некоторые стандартные поля заголовка доступны напрямую через методы вроде `getDate()` и `getContentType()`.

В следующем примере с применением метода `openConnection()` класса `URL` создается объект `URLConnection`, который затем используется для просмотра свойств и содержимого документа:

```
// Демонстрация использования класса URLConnection.

import java.net.*;
import java.io.*;
import java.util.Date;

class UCDemo
{
  public static void main(String[] args) throws Exception {
    int c;
    URL hp = new URL("http://www.internic.net");
    URLConnection hpCon = hp.openConnection();

    // Получить дату.
    long d = hpCon.getDate();
    if(d==0)
      System.out.println("Информация о дате недоступна.");
    else
      System.out.println("Дата: " + new Date(d));

    // Получить тип содержимого.
    System.out.println("Content-Type: " + hpCon.getContentType());

    // Получить дату срока действия.
    d = hpCon.getExpiration();
    if(d==0)
      System.out.println("Информация о сроке действия недоступна.");
```

```
      else
        System.out.println("Срок действия: " + new Date(d));

      // Получить дату последней модификации.
      d = hpCon.getLastModified();
      if(d==0)
        System.out.println("Информация о последней модификации недоступна.");
      else
        System.out.println("Last-Modified: " + new Date(d));

      // Получить длину содержимого.
      long len = hpCon.getContentLengthLong();
      if(len == -1)
        System.out.println("Длина содержимого недоступна.");
      else
        System.out.println("Content-Length: " + len);

      if(len != 0) {
        System.out.println("=== Содержимое ===");
        InputStream input = hpCon.getInputStream();
        while (((c = input.read()) != -1)) {
          System.out.print((char) c);
        }
        input.close();
      } else {
        System.out.println("Содержимое недоступно.");
      }
    }
  }
```

В программе устанавливается подключение HTTP к `www.internic.net` через порт 80. Затем отображаются значения заголовка и извлекается содержимое. Возможно, вам будет интересно поэкспериментировать с этим примером, наблюдая за результатами, и для сравнения опробовать разные вебсайты.

HttpURLConnection

В Java доступен подкласс класса `URLConnection` по имени `HttpURLConnection`, который обеспечивает поддержку подключений HTTP. Объект `HttpURLConnection` получается только что показанным способом, предусматривающим вызов метода `openConnection()` на объекте `URL`, но результат потребуется привести к типу `HttpURLConnection`. (Разумеется, необходимо убедиться в том, что подключение HTTP действительно открыто.) Получив ссылку на объект `HttpURLConnection`, можно применять любой метод, унаследованный от класса `URLConnection`. Вдобавок можно использовать любой из нескольких методов, определенных в `HttpURLConnection`, которые кратко описаны в табл. 24.6.

Таблица 24.6. Избранные методы, определенные в классе HttpURLConnection

Метод	Описание
`static boolean getFollowRedirects()`	Возвращает `true`, если перенаправление осуществляется автоматически, или `false` в противном случае. Автоматическое перенаправление принято по умолчанию
`String getRequestMethod()`	Возвращает строку, представляющую метод выполнения запросов URL. По умолчанию принят метод GET. Доступны и другие варианты, такие как POST
`int getResponseCode() throws IOException`	Возвращает код ответа HTTP. Если код запроса не может быть получен, тогда возвращается –1. В случае разрыва подключения генерируется исключение `IOException`
`String getResponseMessage() throws IOException`	Возвращает сообщение ответа, ассоциированное с кодом ответа, или `null`, если сообщение недоступно. В случае разрыва подключения генерируется исключение `IOException`
`static void setFollowRedirects (boolean how)`	Если в how передается `true`, то перенаправление осуществляется автоматически, а если `false`, тогда перенаправление автоматически не выполняется. Автоматическое перенаправление принято по умолчанию
`void setRequestMethod (String how) throws ProtocolException`	Устанавливает метод выполнения запросов HTTP в тот, что указан в how. Стандартным методом является GET, но доступны и другие варианты вроде POST. Если в how передается недопустимое значение, тогда генерируется исключение `ProtocolException`

В следующей программе демонстрируется применение класса `HttpURLConnection`. Сначала устанавливается подключение к `www.google.com`. Затем выводится метод запроса, код ответа и ответное сообщение. Наконец, отображаются ключи и значения в заголовке ответа.

```
// Демонстрация использования класса HttpURLConnection.

import java.net.*;
import java.io.*;
import java.util.*;

class HttpURLDemo
{
  public static void main(String[] args) throws Exception {
    URL hp = new URL(http://www.google.com");
```

```
    HttpURLConnection hpCon = (HttpURLConnection) hp.openConnection();

    // Отобразить метод запроса.
    System.out.println("Метод запроса: " +
                       hpCon.getRequestMethod());

    // Отобразить код ответа.
    System.out.println("Код ответа: " +
                       hpCon.getResponseCode());

    // Отобразить сообщение ответа.
    System.out.println("Сообщение ответа: " +
                       hpCon.getResponseMessage());

    // Получить список полей заголовка и набор ключей заголовка.
    Map<String, List<String>> hdrMap = hpCon.getHeaderFields();
    Set<String> hdrField = hdrMap.keySet();

    System.out.println("\nЗаголовок:");

    // Отобразить все ключи и значения заголовка.
    for(String k : hdrField) {
      System.out.println("  Ключ: " + k +
                         "  Значение: " + hdrMap.get(k));
    }
  }
}
```

Ниже приведен небольшой фрагмент вывода, выдаваемого программой. (Разумеется, точный ответ, возвращаемый `www.google.com`, со временем будет варьироваться.)

```
Метод запроса: GET
Код ответа: 200
Сообщение ответа: OK

Заголовок:
  Ключ: Transfer-Encoding  Значение: [chunked]
  Ключ: null  Значение: [HTTP/1.1 200 OK]
  Ключ: Server  Значение: [gws]
```

Обратите внимание на способ отображения ключей и значений заголовка. Первым делом получается карта ключей и значений заголовка вызовом метода `getHeaderFields()`, унаследованного от `URLConnection`. Затем извлекается набор ключей заголовка с помощью вызова метода `keySet()` на карте. Далее организуется проход по набору ключей в цикле `for` стиля "for-each". Значение, ассоциированное с каждым ключом, извлекается вызовом метода `get()` на карте.

Класс URI

Класс `URI` инкапсулирует *универсальный идентификатор ресурса* (Uniform Resource Identifier — URI), который похож на URL. Фактически URL являются подмножеством URI. Идентификаторы URI представляет собой стандартный способ идентификации ресурса, а URL также описывает способ получения доступа к ресурсу.

Cookie-наборы

В состав пакета java.net входят классы и интерфейсы, которые помогают управлять cookie-наборами и могут использоваться для создания сеанса HTTP с запоминанием состояния (как противоположность отсутствию отслеживания состояния). Речь идет о классах CookieHandler, CookieManager и HttpCookie, а также об интерфейсах CookiePolicy и CookieStore. Создание сеанса HTTP с запоминанием состояния в книге не рассматривается.

На заметку! Информацию о работе с cookie-наборами и сервлетами ищите в главе 36.

Серверные сокеты TCP/IP

Как упоминалось ранее, в Java есть другой сокетный класс, который необходимо применять для создания серверных приложений. Класс ServerSocket используется для создания серверов, которые прослушивают запросы на подключение к ним через опубликованные порты, поступающие от локальных или удаленных клиентских программ. Серверные сокеты довольно значительно отличаются от обычных сокетов. Созданный объект ServerSocket регистрируется в системе как заинтересованный в клиентских подключениях. Конструкторы класса ServerSocket получают номер порта, на который нужно принимать соединения, и необязательно длину очереди для указанного порта. Длина очереди сообщает системе, сколько клиентских соединений она может оставлять в состоянии ожидании, прежде чем просто отклонять запросы на подключение. Стандартное значение равно 50. При неблагоприятных условиях конструкторы могут генерировать исключение IOException. Три конструктора класса ServerSocket кратко описаны в табл. 24.7.

Таблица 24.7. Конструкторы класса ServerSocket

Конструктор	Описание
`ServerSocket(int port) throws IOException`	Создает серверный сокет на указанном порту с длиной очереди 50
`ServerSocket(int port, int maxQueue) throws IOException`	Создает серверный сокет на указанном порту с максимальной длиной очереди maxQueue
`ServerSocket(int port, int maxQueue, InetAddress localAddress) throws IOException`	Создает серверный сокет на указанном порту с максимальной длиной очереди maxQueue. На групповом хосте в localAddress указывается IP-адрес, к которому привязывается данный сокет

В классе ServerSocket определен метод accept(), представляющий собой блокирующий вызов, который будет ожидать, пока клиент инициирует связь, а затем возвратит обычный сокет, впоследствии применяемый для связи с клиентом.

Дейтаграммы

Сеть в стиле TCP/IP подходит для большинства сетевых потребностей. Она обеспечивает сериализированный, предсказуемый и надежный поток пакетных данных. Однако сетевой обмен не обходится без накладных расходов. Протокол TCP включает в себя множество сложных алгоритмов для управления перегрузкой в насыщенных сетях, а также пессимистичные предположения относительно потери пакетов, что приводит к несколько неэффективному способу передачи данных. Дейтаграммы предлагают альтернативный способ.

Дейтаграммы — это пакеты информации, передаваемые между машинами. Они чем-то напоминают ситуацию, когда играющий вслепую вратарь бьет по мячу в направлении ворот противника. После того как дейтаграмма передана намеченной цели, нет никакой гарантии, что она прибудет или даже что кто-то там ее перехватит. Точно так же, когда дейтаграмма получена, нет никакой гарантии, что она не была повреждена при передаче или что тот, кто ее отправил, все еще находится на месте, чтобы получить ответ.

Дейтаграммы в Java реализованы поверх протокола UDP с использованием двух классов: класса `DatagramPacket`, реализующего контейнер данных, и класса `DatagramSocket`, реализующего механизм для отправки или получения пакетов дейтаграмм. Оба класса исследуются далее в главе.

DatagramSocket

В классе `DatagramSocket` определены четыре открытых конструктора:

```
DatagramSocket() throws SocketException
DatagramSocket(int port) throws SocketException
DatagramSocket(int port, InetAddress ipAddress) throws SocketException
DatagramSocket(SocketAddress address) throws SocketException
```

Первая форма конструктора создает объект `DatagramSocket`, привязанный к любому неиспользуемому порту на локальном компьютере. Вторая форма конструктора создает объект `DatagramSocket`, который привязан к порту, указанному в параметре `port`. Третья форма конструктора создает объект `DatagramSocket`, привязанный к указанному порту и адресу `InetAddress`. Четвертая форма конструктора создает объект `DatagramSocket`, который привязан к адресу, указанному в параметре `address` типа `SocketAddress`. Типом `SocketAddress` является абстрактный класс, который реализован конкретным классом `InetSocketAddress`, инкапсулирующим IP-адрес и номер порта. Все конструкторы будут генерировать исключение `SocketException`, если при создании сокета возникнет ошибка.

В классе `DatagramSocket` определено много методов. Двумя наиболее важными методами следует считать `send()` и `receive()`:

```
void send(DatagramPacket packet) throws IOException
void receive(DatagramPacket packet) throws IOException
```

Метод send() отправляет в порт пакет, указанный в package. Метод receive() ожидает получения пакета и возвращает результат.

В DatagramSocket также определен метод close(), который закрывает сокет. Класс DatagramSocket реализует интерфейс AutoCloseable, т.е. объектом DatagramSocket можно управлять с помощью блока try с ресурсами.

Остальные методы обеспечивают доступ к разнообразным атрибутам, связанным с DatagramSocket. В табл. 24.8 кратко описаны избранные методы.

Таблица 24.8. Избранные методы, определенные в классе DatagramSocket

Метод	Описание
InetAddress getInetAddress()	Если сокет подключен, тогда возвращает адрес, или null в противном случае
int getLocalPort()	Возвращает номер локального порта
int getPort()	Возвращает номер порта, подключенного к сокету. Если сокет не подключен к порту, тогда возвращает -1
boolean isBound()	Возвращает true, если сокет привязан к адресу, или false в противном случае
boolean isConnected()	Возвращает true, если сокет подключен к серверу, или false в противном случае
void setSoTimeout (int millis) throws SocketException	Устанавливает период тайм-аута равным переданному в millis количеству миллисекунд

DatagramPacket

В классе DatagramPacket определено несколько конструкторов, четыре из которых показаны ниже:

```
DatagramPacket(byte[] data, int size)
DatagramPacket(byte[] data, int offset, int size)
DatagramPacket(byte[] data, int size, InetAddress ipAddress, int port)
DatagramPacket(byte[] data, int offset, int size,
   InetAddress ipAddress, int port)
```

В первой форме конструктора указывается буфер, который будет принимать данные (data), и размер пакета (size). Он применяется для получения данных через DatagramSocket. Вторая форма конструктора позволяет указать смещение в буфере, по которому будут сохраняться данные. В третьей форме конструктора задается целевой адрес (ipAddress) и порт (port), которые используются DatagramSocket для определения того, куда будут отправлены данные в пакете. Четвертая форма конструктора обеспечивает передачу пакетов, начиная с указанного в offset смещения внутри data. Считайте, что первые две формы строят "ящик для входящей почты", а вторые две формы заполняют конверт и указывают адрес.

В классе DatagramPacket определено несколько методов, в том числе кратко описанные в табл. 24.9, которые предоставляют доступ к адресу и номеру порта пакета, а также к низкоуровневым данным и их длине.

Таблица 24.9. Избранные методы, определенные в классе DatagramPacket

Метод	Описание
`InetAddress getAddress()`	Возвращает адрес источника (для принимаемых дейтаграмм) или места назначения (для отправляемых дейтаграмм)
`byte[] getData()`	Возвращает байтовый массив данных, содержащихся в дейтаграмме. В основном используется для извлечения данных из дейтаграммы после ее получения
`int getLength()`	Возвращает длину допустимых данных, содержащихся в байтовом массиве, который будет возвращен из метода `getData()`. Значение может не быть равно длине всего байтового массива
`int getOffset()`	Возвращает начальный индекс данных
`int getPort()`	Возвращает номер порта
`void setAddress (InetAddress ipAddress)`	Устанавливает адрес, по которому пакет будет отправлен. Адрес указывается в `ipAddress`
`void setData(byte[] data)`	Устанавливает данные в `data`, смещение в 0 и длину в количество байтов в `data`
`void setData (byte[] data, int idx, int size)`	Устанавливает данные в `data`, смещение в `idx` и длину в `size`
`void setLength (int size)`	Устанавливает длину пакета в `size`
`void setPort(int port)`	Устанавливает порт в `port`

Пример использования дейтаграмм

В следующем примере реализовано очень простой сетевое взаимодействие клиента и сервера. Сообщения набираются в окне на сервере и передаются по сети на сторону клиента, где они отображаются.

```
// Демонстрация использования дейтаграмм.

import java.net.*;

class WriteServer {
  public static int serverPort = 998;
  public static int clientPort = 999;
```

```
public static int buffer_size = 1024;
public static DatagramSocket ds;
public static byte[] buffer = new byte[buffer_size];
public static void TheServer() throws Exception {
  int pos=0;
  while (true) {
    int c = System.in.read();
    switch (c) {
      case -1:
        System.out.println("Сервер завершает сеанс связи.");
        ds.close();
        return;
      case '\r':
        break;
      case '\n':
        ds.send(new DatagramPacket(buffer,pos,
              InetAddress.getLocalHost(),clientPort));
        pos=0;
        break;
      default:
        buffer[pos++] = (byte) c;
    }
  }
}
public static void TheClient() throws Exception {
  while(true) {
    DatagramPacket p = new DatagramPacket(buffer, buffer.length);
    ds.receive(p);
    System.out.println(new String(p.getData(), 0, p.getLength()));
  }
}
public static void main(String[] args) throws Exception {
  if(args.length == 1) {
    ds = new DatagramSocket(serverPort);
    TheServer();
  } else {
    ds = new DatagramSocket(clientPort);
    TheClient();
  }
}
```

С помощью конструктора `DatagramSocket` программа ограничивается работой между двумя портами на локальной машине. Для использования программы введите в одном окне показанную ниже команду, которая запустит клиент:

```
java WriteServer
```

Затем введите команду, которая запустит сервер:

```
java WriteServer 1
```

Все, что набирается в окне сервера, будет отправлено в окно клиента после получения символа новой строки.

На заметку! Применение дейтаграмм на вашем компьютере может быть запрещено. (Скажем, брандмауэр способен препятствовать их использованию.) В таком случае предыдущий пример запустить не удастся. Кроме того, номера портов, указанные в программе, возможно, придется скорректировать для имеющейся в наличии среды.

Введение в пакет `java.net.http`

В предыдущем материале была представлена традиционная поддержка Java для работы в сети, обеспечиваемая пакетом `java.net`. Этот API-интерфейс доступен во всех версиях Java и применяется весьма широко. Таким образом, знание традиционного подхода Java к работе в сети важно для всех программистов. Тем не менее, начиная с версии JDK 11, в модуль `java.net.http` был добавлен новый пакет по имени `java.net.http`, который предлагает расширенную и обновленную сетевую поддержку для клиентов HTTP. Упомянутый новый API-интерфейс обычно называют *HTTP Client API*.

Для многих видов сетей HTTP возможности, определенные API-интерфейсом в `java.net.http`, могут обеспечить превосходные решения. Помимо оптимизированного и простого в использовании API-интерфейса к другим преимуществам относятся поддержка асинхронной связи, HTTP/2 и управление потоком. В целом HTTP Client API разработан в качестве более совершенной альтернативы функциональности класса `HttpURLConnection`. Он также поддерживает протокол WebSocket для двунаправленной связи.

Ниже обсуждается ряд ключевых средств HTTP Client API. Имейте в виду, что его возможности гораздо шире, чем описано в главе. Если вы планируете разрабатывать сложный сетевой код, то должны как следует изучить пакет `java.net.http`. Цель здесь состоит в том, чтобы предложить введение в некоторые основы.

Три ключевых элемента

Внимание в последующем обсуждении сосредоточено главным образом на трех основных элементах HTTP Client API, которые перечислены в табл. 24.10.

Таблица 24.10. Три ключевых элемента HTTP Client API

Класс	Описание
`HttpClient`	Инкапсулирует клиент HTTP. Предоставляет средства, с помощью которых будет отправляться запрос и получаться ответ
`HttpRequest`	Инкапсулирует запрос
`HttpResponse`	Инкапсулирует ответ

Они работают вместе для поддержки функциональных средств, относящихся к запросам и ответам HTTP. Рассмотрим общую процедуру. Сначала необходимо создать объект `HttpClient`, затем объект `HttpRequest` и отправить его, вызвав метод `send()` на объекте `HttpClient`. Ответ возвращает ме-

тод send(). Из ответа можно получить заголовки и тело ответа. Прежде чем проработать пример, имеет смысл ознакомиться с обзором этих фундаментальных аспектов HTTP Client API.

HttpClient

Класс HttpClient инкапсулирует механизм запросов и ответов HTTP. Он поддерживает как синхронную, так и асинхронную связь. В книге будет применяться только синхронная связь, но вы можете поэкспериментировать с асинхронной связью самостоятельно. После получения объект HttpClient можно использовать для отправки запросов и получения ответов. Таким образом, он лежит в основе HTTP Client API.

Класс HttpClient является абстрактным, так что его экземпляры не создаются с помощью открытого конструктора. Взамен для создания экземпляров будет применяться фабричный метод. Класс HttpClient поддерживает *построители* через интерфейс HttpClient.Builder, который предлагает несколько методов, позволяющих конфигурировать объект реализации HttpClient. Для получения построителя HttpClient используется статический метод newBuilder(). Он возвращает построитель, который позволяет конфигурировать создаваемый объект реализации HttpClient. Вызов метода build() на построителе приводит к созданию и возвращению экземпляра реализации HttpClient. Например, следующий код создает объект реализации HttpClient со стандартными настройками:

```
HttpClient myHC = HttpClient.newBuilder().build();
```

В интерфейсе HttpClient.Builder определено несколько методов, позволяющих конфигурировать построитель. Вот один пример. По умолчанию перенаправления не выполняются. Ситуацию можно изменить, вызвав метод followRedirects() и передав ему новый параметр перенаправления, который должен быть значением перечисления HttpClient.Redirect: ALWAYS, NEVER или NORMAL. Первые два, ALWAYS и NEVER, говорят сами за себя, т.е. осуществлять перенаправление всегда или никогда. Параметр NORMAL обеспечивает выполнение перенаправления, если только оно не производится с сайта HTTPS на сайт HTTP. Например, показанный ниже код создает построитель с политикой перенаправления NORMAL. Затем данный построитель применяется для создания объекта реализации HttpClient.

```
HttpClient.Builder myBuilder =
  HttpClient.newBuilder().followRedirects(HttpClient.Redirect.NORMAL);
HttpClient.myHC = myBuilder.build();
```

Помимо прочего параметры конфигурации построителя включают аутентификацию, прокси-сервер, версию HTTP и приоритет. Таким образом, клиент HTTP можно создавать практически для любых нужд.

В случаях, когда стандартной конфигурации оказывается достаточно, объект реализации HttpClient по умолчанию можно получить напрямую, вызвав метод newHttpClient():

```
static HttpClient newHttpClient()
```

Метод возвращает объект реализации HttpClient со стандартной конфигурацией. Скажем, следующий оператор создает новый стандартный объект реализации HttpClient:

```
HttpClient myHC = HttpClient.newHttpClient();
```

Поскольку для целей настоящей книги вполне достаточно стандартного клиента, именно такой подход задействован в будущих примерах.

После получения экземпляра реализации HttpClient можно отправить синхронный запрос, вызвав его метод send():

```
<T> HttpResponse <T> send(HttpRequest req, HttpResponse.BodyHandler<T>
  handler) throws IOException, InterruptedException
```

В req указывается запрос, а в handler — способ обработки тела ответа. Как вскоре вы увидите, часто можно использовать один из предопределенных обработчиков тела, предоставляемых классом HttpResponse.BodyHandlers. Метод send() возвращает объект HttpResponse. Таким образом, send() обеспечивает базовый механизм связи по протоколу HTTP.

HttpRequest

Запросы в HTTP Client API инкапсулируются в абстрактном классе HttpRequest. Для создания объекта реализации HttpRequest будет применяться построитель. Чтобы получить построитель, понадобится вызвать метод newBuilder() класса HttpRequest. Вот две его формы:

```
static HttpRequest.Builder newBuilder()
static HttpRequest.Builder newBuilder(URI uri)
```

Первая форма метода newBuilder() создает стандартный построитель. Вторая форма разрешает указывать URI ресурса. Существует также третья форма, позволяющая получить построитель, который может создавать объект HttpRequest, похожий на указанный объект HttpRequest.

Интерфейс HttpRequest.Builder позволяет задавать разнообразные аспекты запроса, например, используемый метод запроса. (По умолчанию применяется GET.) Кроме того, можно устанавливать информацию заголовка, URI, версию HTTP и т.д. Помимо URI стандартных настроек часто оказывается достаточно. Вызвав метод method() на объекте HttpRequest, можно получить строковое представление метода запроса.

Чтобы фактически создать запрос, необходимо вызвать метод build() на экземпляре построителя:

```
HttpRequest build()
```

Имеющийся экземпляр HttpRequest можно использовать в вызове метода send() объекта реализации HttpClient, как было показано в предыдущем разделе.

HttpResponse

Ответы в HTTP Client API инкапсулируются в реализации интерфейса HttpResponse — абстрактного интерфейса со следующим объявлением:

```
HttpResponse<T>
```

В T указывается тип тела. Поскольку тип тела является обобщенным, он позволяет обрабатывать тело различными способами, что дает большую степень гибкости в написании кода ответа.

При отправке запроса возвращается экземпляр реализации HttpResponse, содержащий ответ. В интерфейсе HttpResponse определено несколько методов, которые предоставляют доступ к информации в ответе. Возможно, наиболее важным из них считается метод body():

```
T body()
```

Метод body() возвращает ссылку на тело. Конкретный тип ссылки определяется типом T, который задается обработчиком тела, указанным с помощью метода send(). Получить код состояния, связанный с ответом, можно вызовом метода statusCode():

```
int statusCode()
```

Метод возвращает код состояния HTTP. Значение 200 отражает успех.

Еще одним методом в HttpResponse является headers(), который получает заголовки ответа:

```
HttpHeaders headers()
```

Заголовки, ассоциированные с ответом, инкапсулируются в экземпляре класса HttpHeaders. В нем определены различные методы, которые обеспечивают доступ к заголовкам. В рассматриваемом далее примере применяется метод map():

```
Map<String, List<String>> map()
```

Он возвращает карту, содержащую все поля и значения заголовков.

Одно из преимуществ HTTP Client API заключается в том, что ответы могут обрабатываться автоматически и разнообразными способами. Ответы обрабатываются реализациями интерфейса HttpResponse.BodyHandler. В классе HttpResponse.BodyHandlers предоставляется несколько предопределенных фабричных методов для обработки тела. В табл. 24.11 приведены три примера.

Другие предопределенные обработчики получают тело ответа в виде байтового массива, потока строк, загружаемого файла и объекта реализации Flow.Publisher. Кроме того, поддерживается потребитель без управления потоком. Прежде чем двигаться дальше, важно отметить, что поток, возвращаемый методом ofInputStream(), должен быть прочитан полностью. Это позволяет освободить ассоциированные с ним ресурсы. Если по какой-либо причине все тело не может быть прочитано, тогда нужно закрыть поток посредством вызова метода close(), что также может повлечь за собой закрытие подключения HTTP. В общем случае лучше всего прочитать весь поток.

Простой пример клиента HTTP

В показанном ниже примере задействованы только что описанные функциональные средства HTTP Client API. В нем демонстрируется отправка запроса, отображение тела ответа и получение списка заголовков ответа.

Таблица 24.11. Три примера фабричных методов для обработки тела, определенные в классе `HttpResponse.BodyHandlers`

Метод	Описание
`static HttpResponse.BodyHandler <Path> ofFile(Path filename)`	Записывает тело ответа в файл, указанный с помощью `filename`. После получения ответа метод `HttpResponse.body()` возвратит объект реализации `Path` для файла
`static HttpResponse.BodyHandler <InputStream> ofInputStream()`	Открывает объект `InputStream` для тела ответа. После получения ответа метод `HttpResponse.body()` возвратит ссылку на объект `InputStream`
`static HttpResponse.BodyHandler <String> ofString()`	Помещает тело ответа в строку. После получения ответа метод `HttpResponse.body()` возвратит строку

Сравните его с похожими фрагментами кода из показанных ранее программ `UCDemo` и `HttpURLDemo`. Обратите внимание, что здесь для получения потока ввода, связанного с телом ответа, применяется метод `ofInputStream()`.

```
// Демонстрация использования HttpClient.
import java.net.*;
import java.net.http.*;
import java.io.*;
import java.util.*;
class HttpClientDemo
{
  public static void main(String[] args) throws Exception {
    // Получить клиент со стандартными параметрами.
    HttpClient myHC = HttpClient.newHttpClient();
    // Создать запрос.
    HttpRequest myReq = HttpRequest.newBuilder(
                        new URI("http://www.google.com/")).build();
    //Отправить запрос и получить ответ.Для тела применяется объект InputStream.
    HttpResponse<InputStream> myResp = myHC.send(myReq,
                        HttpResponse.BodyHandlers.ofInputStream());
    // Отобразить код ответа и метод запроса.
    System.out.println("Код ответа: " + myResp.statusCode());
    System.out.println("Метод запроса: " + myReq.method());
    // Получить заголовок из ответа.
    HttpHeaders hdrs = myResp.headers();
    // Получить карту с полями заголовка.
    Map<String, List<String>> hdrMap = hdrs.map();
    Set<String> hdrField = hdrMap.keySet();
    System.out.println("\nЗаголовок:");
    // Отобразить все ключи и значения заголовка.
    for(String k : hdrField) {
      System.out.println("  Ключ: " + k + "  Значение: " + hdrMap.get(k));
    }
```

```
    // Отобразить тело.
    System.out.println("\nТело: ");

    InputStream input = myResp.body();
    int c;
    // Прочитать и отобразить все тело.
    while((c = input.read()) != -1) {
      System.out.print((char) c);
    }
  }
}
```

В программе сначала создается объект реализации `HttpClient`, который затем служит для отправки запроса веб-сайту `www.google.com` (конечно, его можно заменить другим желаемым веб-сайтом). Обработчик тела использует входной поток через `ofInputStream()`. Далее отображается код состояния ответа и метод запроса, после чего заголовок и тело. Поскольку метод `ofInputStream()` был указан в методе `send()`, метод `body()` возвратит объект `InputStream`, который применяется для чтения и отображения тела.

Для обработки тела в предыдущей программе использовался поток ввода в целях сравнения с показанной ранее программой `UCDemo`, где применялся похожий подход. Однако доступны и другие варианты. Скажем, можно воспользоваться методом `ofString()` для обработки тела как строки. При таком подходе после получения ответа тело будет находиться в экземпляре `String`. Чтобы опробовать его, модифицируйте строку с вызовом `send()` следующим образом:

```
HttpResponse<InputStream> myResp = myHC.send(myReq,
  HttpResponse.BodyHandlers.ofString());
```

Далее замените код чтения и отображения тела с применением потока ввода:

```
System.out.println(myResp.body());
```

Так как тело ответа уже хранится в строке, его можно вывести напрямую. Возможно, вы решите поэкспериментировать и с другими обработчиками тела. Особый интерес представляет метод `ofLines()`, который позволяет обращаться к телу как к потоку строк. Одно из преимуществ HTPP Client API как раз и связано с наличием встроенных обработчиков тела для различных ситуаций.

Что еще рекомендуется изучить в `java.net.http`

В приведенном выше введении был описан ряд основных средств HTPP Client API в пакете `java.net.http`, но есть еще несколько, которые рекомендуется исследовать. Одним из наиболее важных средств следует считать класс `WebSocket`, поддерживающий двустороннюю связь, а другим — асинхронность, поддерживаемая API-интерфейсом. Вообще говоря, если вы планируете заниматься сетевым программированием, тогда должны тщательно изучить пакет `java.net.http`. Он является важным дополнением к API-интерфейсам Java, предназначенным для организации работы в сети.

ГЛАВА 25 Обработка событий

В этой главе рассматривается важный аспект Java: события. Обработка событий играет основополагающую роль для программирования на Java, поскольку является неотъемлемой частью создания многочисленных видов приложений. Например, любая программа, использующая графический пользовательский интерфейс, такая как приложение Java, написанное для Windows, управляется именно событиями. Таким образом, разрабатывать программы такого типа без надежного владения обработкой событий не удастся. События поддерживаются несколькими пакетами, включая `java.util`, `java.awt` и `java.awt.event`.

Начиная с версии JDK 9, пакеты `java.awt` и `java.awt.event` входят в состав модуля `java.desktop`, а `java.util` находятся в модуле `java.base`.

Многие события, на которые будет реагировать ваша программа, генерируются во время взаимодействия пользователя с программой на базе графического пользовательского интерфейса. Как раз такие типы событий и исследуются в настоящей главе. Они передаются в программу различными способами, причем конкретный метод зависит от фактического события. Существует несколько типов событий, среди которых события, генерируемые мышью, клавиатурой и различными элементами управления графического интерфейса вроде кнопки, линейки прокрутки или флажка.

Глава начинается с обзора механизма обработки событий Java. Затем обсуждается ряд классов событий и интерфейсов из библиотеки AWT (Abstract Window Toolkit), которая была первой инфраструктурой для построения графических пользовательских интерфейсов Java и позволяет довольно просто представить основы обработки событий. Затем в главе приводится несколько примеров, демонстрирующих основы обработки событий. Кроме того, в главе вводятся ключевые понятия, связанные с программированием для графических пользовательских интерфейсов, и объясняется, как применять классы адаптеров, внутренние классы и анонимные внутренние классы с целью оптимизации кода, обрабатывающего события. Такие методики часто используются в примерах, представленных далее в книге.

На заметку! Основное внимание в главе уделяется событиям, связанным с программами на базе графических пользовательских интерфейсов. Однако события иногда применяются и для целей, не связанных напрямую с программами подобного рода. Во всех случаях применяются одни и те же способы обработки событий.

Два механизма обработки событий

Прежде чем приступить к обсуждению обработки событий, необходимо сделать важное замечание: способ обработки событий, принятый в исходной версии Java (1.0), значительно изменился во всех последующих версиях Java, начиная с 1.1. Хотя метод обработки событий версии 1.0 по-прежнему поддерживается, использовать его в новых программах не рекомендуется. Кроме того, многие методы, поддерживающие старую модель событий версии 1.0, были объявлены нерекомендуемыми. Современный подход предусматривает обработку событий во всех новых программах так, как описано в настоящей главе.

Модель делегирования обработки событий

Современный подход к обработке событий основан на *модели делегирования обработки событий*, которая определяет стандартные и согласованные механизмы генерации и обработки событий. Его концепция довольно проста: *источник* генерирует событие и отправляет его одному или нескольким прослушивателям. В данной схеме прослушиватель просто ожидает до тех пор, пока не получит событие. Как только событие получено, прослушиватель обрабатывает его и возвращает управление. Преимущество такой схемы связано с тем, что логика приложения, которая обрабатывает события, четко отделена от логики пользовательского интерфейса, генерирующей эти события. Элемент пользовательского интерфейса способен "делегировать" обработку события отдельному фрагменту кода.

В модели делегирования обработки событий прослушиватели должны регистрироваться в источнике, чтобы получать уведомления о поступлении событий, что обеспечивает важное преимущество: уведомления отправляются только тем прослушивателям, которые желают их получать. Такой способ обработки событий гораздо эффективнее схемы, которая применялась в исходном подходе Java 1.0, когда событие распространялось вверх по иерархии включения до тех пор, пока его не обрабатывал какой-нибудь компонент. В итоге компоненты должны были получать события, которые они не обрабатывали, и понапрасну тратилось время. Модель делегирования обработки событий устраняет накладные расходы подобного рода.

В последующих разделах определяются события и описываются роли источников и прослушивателей.

События

В модели делегирования *событие* представляет собой объект, описывающий изменение состояния источника. Среди других причин событие может быть сгенерировано как следствие взаимодействия человека с элементами графического пользовательского интерфейса. Действия, которые вызывают генерацию событий, включают нажатие кнопки, ввод символа с клавиатуры, выбор элемента в списке и щелчок кнопкой мыши. В качестве примеров можно привести множество других пользовательских операций.

Также могут происходить события, которые напрямую не связаны с взаимодействием с пользовательским интерфейсом. Например, событие может генерироваться, когда истекает время таймера, значение счетчика превышает установленное значение, возникает программный или аппаратный сбой или операция завершается. Можно свободно определять события, подходящие для разрабатываемого приложения.

Источники событий

Источник представляет собой объект, который генерирует событие. Это происходит, когда внутреннее состояние данного объекта каким-либо образом изменяется. Источники могут генерировать более одного типа событий.

Источник должен зарегистрировать прослушиватели, чтобы они могли получать уведомления о событиях конкретного типа. Для каждого типа события предусмотрен свой метод регистрации. Вот общая форма:

```
public void addTypeListener (TypeListener el)
```

В `Type` указывается имя события, а в `el` — ссылка на прослушиватель событий. Например, метод, который регистрирует прослушиватель событий клавиатуры, называется `addKeyListener()`, а метод, регистрирующий прослушиватель движения мыши, имеет имя `addMouseMotionListener()`. Когда происходит событие, все зарегистрированные прослушиватели уведомляются и получают копию объекта события, что называется *групповой рассылкой* события. Во всех случаях уведомления отправляются только тем прослушивателям, которые зарегистрировались для их получения.

Некоторые источники могут допускать регистрацию только одного прослушивателя. Ниже показана общая форма метода подобного рода:

```
public void addTypeListener(TypeListener el)
  throws java.util.TooManyListenersException
```

В `Type` передается имя события, а в `el` — ссылка на прослушиватель событий. Когда такое событие происходит, зарегистрированный прослушиватель уведомляется, что называется *индивидуальной рассылкой* события.

Источник также обязан предоставлять метод, который позволяет прослушивателю отменять регистрацию для события конкретного типа и имеет следующую общую форму:

```
public void removeTypeListener(TypeListener el)
```

В Type указывается имя события, а в el — ссылка на прослушиватель событий. Скажем, чтобы удалить прослушиватель событий клавиатуры, потребуется вызвать метод removeKeyListener().

Методы для добавления и удаления прослушивателей предоставляются источником, генерирующим события. Например, Component, который является классом верхнего уровня, определенным в AWT, предлагает методы добавления и удаления прослушивателей событий клавиатуры и мыши.

Прослушиватели событий

Прослушиватель — это объект, который уведомляется при возникновении события. К нему предъявляются два основных требования. Во-первых, он должен быть зарегистрирован в одном или нескольких источниках для получения уведомлений о специфических типах событий. Во-вторых, он должен реализовывать методы для получения и обработки таких уведомлений. Другими словами, прослушиватель должен предоставлять обработчики событий.

Методы, которые получают и обрабатывают события, определены в наборе интерфейсов, таком как находящийся в пакете java.awt.event. Скажем, в интерфейсе MouseMotionListener определены два метода для получения уведомлений при перетаскивании или перемещении указателя мыши. Любой объект может обрабатывать одно или оба таких события, если он предоставляет реализацию упомянутого интерфейса. Другие интерфейсы прослушивателя обсуждаются позже в текущей и последующих главах.

С событиями связан еще один ключевой момент: обработчик события должен быстро возвращать управление. По большей части программа с графическим пользовательским интерфейсом не должна переходить в "режим" работы, в котором она сохраняет управление в течение длительного периода времени. Взамен такая программа обязана выполнять определенные действия в ответ на события, а затем возвращать управление исполняющей среде. Отказ от удовлетворения такого требования может привести к тому, что программа будет работать медленно или даже не реагировать на запросы. Если программе необходимо выполнить повторяющуюся задачу, например, прокрутку баннера, тогда она должна запускать отдельный поток. Короче говоря, когда программа получает событие, она обязана немедленно его обработать, а затем возвратить управление.

Классы событий

В основе механизма обработки событий Java лежат классы, представляющие события. Таким образом, обсуждение обработки событий следует начинать с классов событий. Тем не менее, важно понимать, что в Java определено множество типов событий и в данной главе удастся обсудить далеко не все классы событий. Вероятно, наиболее широко используемыми событиями на момент написания книги были события, определенные в AWT и Swing.

Основное внимание в главе уделяется событиям из AWT, большая часть которых применима и к Swing. Некоторые специфичные для Swing события описаны в главе 32 при рассмотрении Swing.

В корне иерархии классов событий Java находится класс `EventObject` из пакета `java.util`. Он является суперклассом для всех событий. Один его конструктор показан ниже:

```
EventObject(Object src)
```

В `src` указывается объект, генерирующий событие.

В классе `EventObject` определены два метода: `getSource()` и `toString()`. Метод `getSource()` возвращает источник события. Вот его общий вид:

```
Object getSource()
```

Метод `toString()` вполне ожидаемо возвращает строковый эквивалент события.

Класс `AWTEvent`, определенный в пакете `java.awt`, представляет собой подкласс `EventObject` и выступает в качестве суперкласса (прямо или косвенно) для всех событий, основанных на AWT, которые используются моделью делегирования обработки событий. Метод `getID()` класса `AWTEvent` можно применять для определения типа события; его сигнатура выглядит следующим образом:

```
int getID()
```

Как правило, функциональные средства, определенные в классе `AWTEvent`, не будут использоваться напрямую. Взамен будут применяться его подклассы. На данный момент важно знать лишь то, что все остальные классы, обсуждаемые в настоящем разделе, являются подклассами `AWTEvent`.

Подведем итоги:

- `EventObject` — суперкласс для всех событий;
- `AWTEvent` — суперкласс для всех событий AWT, обрабатываемых посредством модели делегирования обработки событий.

В пакете `java.awt.event` определено множество типов событий, которые генерируются разнообразными элементами пользовательского интерфейса. В табл. 25.1 кратко описаны часто используемые классы событий и обстоятельства, когда они генерируются. Распространенные конструкторы и методы каждого класса обсуждаются в последующих разделах.

Класс `ActionEvent`

Событие типа `ActionEvent` генерируется при щелчке на кнопке, двойном щелчке на элементе списка или выборе пункта меню. В классе `ActionEvent` определены четыре целочисленные константы, которые можно применять для идентификации любых модификаторов, связанных с событием действия: `ALT_MASK`, `CTRL_MASK`, `META_MASK` и `SHIFT_MASK`.

Таблица 25.1. Часто используемые классы событий в пакете `java.awt.event`

Класс	Описание
`ActionEvent`	Генерируется при нажатии кнопки, двойном щелчке на элементе списка или выборе пункта меню
`AdjustmentEvent`	Генерируется при манипулировании полосой прокрутки
`ComponentEvent`	Генерируется, когда компонент скрывается, перемещается, изменяет размер или становится видимым
`ContainerEvent`	Генерируется, когда компонент добавляется в контейнер или удаляется из него
`FocusEvent`	Генерируется, когда компонент получает или утрачивает фокус ввода с клавиатуры
`InputEvent`	Абстрактный суперкласс для всех классов событий, связанных с вводом в компонентах
`ItemEvent`	Генерируется, когда совершается щелчок на флажке или элементе списка; также возникает при выборе варианта либо выборе или отмене выбора отмечаемого пункта меню
`KeyEvent`	Генерируется при получении ввода с клавиатуры
`MouseEvent`	Генерируется при перетаскивании и перемещении указателя мыши, а также при щелчке, нажатии и отпускании кнопок мыши; также возникает, когда указатель мыши наводится на компонент или покидает его
`MouseWheelEvent`	Генерируется при прокручивании колесика мыши
`TextEvent`	Генерируется при изменении значения в текстовой области или текстовом поле
`WindowEvent`	Генерируется, когда окно становится активным, закрывается, перестает быть активным, разворачивается, сворачивается, открывается или закрывается

Вдобавок существует целочисленная константа `ACTION_PERFORMED`, с помощью которой можно идентифицировать события действий.

Класс `ActionEvent` имеет три конструктора:

```
ActionEvent(Object src, int type, String cmd)
ActionEvent(Object src, int type, String cmd, int modifiers)
ActionEvent(Object src, int type, String cmd, long when, int modifiers)
```

В `src` передается ссылка на объект, сгенерировавший событие. Тип события задается в `type`, а его строка команды — в `cmd`. В `modifiers` указывается,

какие модифицирующие клавиши (<Alt>, <Ctrl>, <Meta> и/или <Shift>) были нажаты, когда сгенерировалось событие. В when указывается, когда произошло событие.

Получить имя команды для вызывающего объекта ActionEvent можно посредством метода getActionCommand():

```
String getActionCommand()
```

Например, при щелчке на кнопке генерируется событие действия, имя команды которого совпадает с меткой на этой кнопке.

Метод getModifiers() возвращает значение, которое отражает, какие модифицирующие клавиши (<Alt>, <Ctrl>, <Meta> и/или <Shift>) были нажаты во время генерации события:

```
int getModifiers()
```

Метод getWhen() возвращает время, когда произошло событие, которое называется *отметкой времени* события:

```
long getWhen()
```

Класс AdjustmentEvent

Событие типа AdjustmentEvent генерируется полосой прокрутки. Существуют пять типов событий корректировки. Для их идентификации предназначены целочисленные константы, определенные в классе AdjustmentEvent, которые кратко описаны в табл. 25.2.

Таблица 25.2. Константы, определенные в классе AdjustmentEvent

Константа	Описание
BLOCK_DECREMENT	Пользователь щелкнул кнопкой мыши внутри полосы прокрутки, чтобы уменьшить связанное с ней значение
BLOCK_INCREMENT	Пользователь щелкнул кнопкой мыши внутри полосы прокрутки, чтобы увеличить связанное с ней значение
TRACK	Пользователь переместил ползунок полосы прокрутки
UNIT_DECREMENT	Пользователь щелкнул кнопкой мыши на кнопке в начале полосы прокрутки, чтобы уменьшить связанное с ней значение
UNIT_INCREMENT	Пользователь щелкнул кнопкой мыши на кнопке в конце полосы прокрутки, чтобы увеличить связанное с ней значение

Кроме того, есть целочисленная константа ADJUSTMENT_VALUE_CHANGED, которая указывает на то, что произошло изменение.

В классе AdjustmentEvent определен один конструктор:

```
AdjustmentEvent(Adjustable src, int id, int type, int val)
```

В src передается ссылка на объект, сгенерировавший событие. В id указывается событие. Тип корректировки задается в type, а связанное с ним значение — в val.

Метод getAdjustable() возвращает объект, сгенерировавший событие, и имеет следующую форму:

```
Adjustable getAdjustable()
```

Тип события корректировки можно получить с помощью метода getAdjustmentType(), который возвращает одну из констант, определенных в AdjustmentEvent:

```
int getAdjustmentType()
```

Величину корректировки можно выяснить посредством метода getValue():

```
int getValue()
```

При манипулировании полосой прокрутки данный метод возвращает значение, представленное таким изменением.

Класс ComponentEvent

Событие типа ComponentEvent генерируется при изменении размера, положения или видимости компонента. Существуют четыре типа событий компонента, для идентификации которых применяются целочисленные константы, определенные в классе ComponentEvent. Эти константы кратко описаны в табл. 25.3.

Таблица 25.3. Константы, определенные в классе ComponentEvent

Константа	Описание
COMPONENT_HIDDEN	Компонент был скрыт
COMPONENT_MOVED	Компонент был перемещен
COMPONENT_RESIZED	Размеры компонента были изменены
COMPONENT_SHOWN	Компонент стал видимым

Класс ComponentEvent имеет следующий конструктор:

```
ComponentEvent(Component src, int type)
```

В src передается ссылка на объект, сгенерировавший событие. Тип события указывается в type.

Класс ComponentEvent прямо или косвенно является суперклассом для классов ContainerEvent, FocusEvent, KeyEvent, MouseEvent и WindowEvent, а также ряда других.

Метод getComponent() возвращает компонент, который сгенерировал событие:

```
Component getComponent()
```

Класс ContainerEvent

Событие типа ContainerEvent генерируется, когда компонент добавляется в контейнер или удаляется из него. Существуют два вида контейнерных событий, для идентификации которых в классе ContainerEvent определены целочисленные константы COMPONENT_ADDED и COMPONENT_REMOVED. Они указывают, что компонент был добавлен в контейнер или удален из него.

Класс ContainerEvent является подклассом ComponentEvent и имеет следующий конструктор:

```
ContainerEvent(Component src, int type, Component comp)
```

В src передается ссылка на контейнер, сгенерировавший событие. Тип события указывается в type, а компонент, который был добавлен в контейнер или удален из него — в comp.

Получить ссылку на контейнер, сгенерировавший событие, можно с использованием метода getContainer():

```
Container getContainer()
```

Метод getChild() возвращает ссылку на компонент, который был добавлен в контейнер или удален из него:

```
Component getChild()
```

Класс FocusEvent

Событие типа FocusEvent генерируется, когда компонент получает или утрачивает фокус ввода. Такие события идентифицируются целочисленными константами FOCUS_GAINED и FOCUS_LOST.

Класс FocusEvent является подклассом ComponentEvent и имеет следующие конструкторы:

```
FocusEvent(Component src, int type)
FocusEvent(Component src, int type, boolean temporaryFlag)
FocusEvent(Component src, int type, boolean temporaryFlag, Component other)
FocusEvent(Component src, int type, boolean temporaryFlag, Component other,
           FocusEvent.Cause what)
```

В src передается ссылка на компонент, сгенерировавший событие. Тип события указывается в type. Параметр temporaryFlag равен true, если событие фокуса является временным, или false в противном случае. (Событие временного фокуса происходит в результате другой операции пользовательского интерфейса. Например, пусть фокус находится в текстовом поле. Если пользователь перемещает указатель мыши для настройки полосы прокрутки, то фокус временно утрачивается.)

В other передается другой компонент, участвующий в смене фокуса, который называется *противоположным компонентом*. Следовательно, если произошло событие FOCUS_GAINED, тогда other будет ссылаться на компонент, утративший фокус. И наоборот, если возникло событие FOCUS_LOST, то other будет ссылаться на компонент, получивший фокус.

Четвертый конструктор был добавлен в версии JDK 9. В его параметре what указывается, почему было сгенерировано событие, в виде значения перечисления FocusEvent.Cause, которое определяет причину события, связанного с фокусом ввода. Перечисление FocusEvent.Cause тоже появилось в JDK 9. Выяснить другой компонент можно, вызвав метод getOppositeComponent():

```
Component getOppositeComponent()
```

Метод возвращает противоположный компонент.

Метод isTemporary() указывает, было ли изменение фокуса временным. Вот его общая форма:

```
boolean isTemporary()
```

Он возвращает true, если изменение временное, или false в противном случае. Начиная с версии JDK 9, можно получить причину возникновения события, вызвав метод getCause():

```
final FocusEvent.Cause getCause()
```

Причина возвращается в виде значения перечисления FocusEvent.Cause.

Класс InputEvent

Абстрактный класс InputEvent является подклассом ComponentEvent и суперклассом для событий ввода в компонентах. Он имеет подклассы KeyEvent и MouseEvent.

В классе InputEvent определено несколько целочисленных констант, представляющих любые модификаторы вроде нажатой управляющей клавиши, которые могут быть связаны с событием. Первоначально в классе InputEvent были определены следующие восемь значений для представления модификаторов, и их все еще можно встретить в унаследованном коде:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

Однако из-за возможных конфликтов между модификаторами, которые применялись для событий клавиатуры и мыши, а также других проблем были добавлены следующие расширенные значения модификаторов:

ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CTRL_DOWN_MASK	

При написании нового кода должны использоваться новые расширенные модификаторы, а не первоначальные. Более того, в версии JDK 9 первоначальные модификаторы объявлены нерекомендуемыми.

Для проверки, была ли нажата модифицирующая клавиша во время генерации события, можно применять методы isAltDown(), isAltGraphDown(), isControlDown(), isMetaDown() и isShiftDown(). Их формы показаны ниже:

```
boolean isAltDown()
boolean isAltGraphDown()
boolean isControlDown()
boolean isMetaDown()
boolean isShiftDown()
```

Вызвав приведенный далее метод `getModifiers()`, можно получить значение, содержащее все первоначальные модификаторы:

```
int getModifiers()
```

Хотя вызовы `getModifiers()` по-прежнему встречаются в унаследованном коде, важно отметить, что из-за объявления первоначальных модификаторов нерекомендуемыми в версии JDK 9 метод `getModifiers()` тоже считается нерекомендуемым, начиная с JDK 9. Взамен нужно получать расширенные модификаторы с помощью метода `getModifiersEx()`:

```
int getModifiersEx()
```

Класс `ItemEvent`

Событие типа `ItemEvent` генерируется при щелчке на флажке или элементе списка либо при выборе или отмене выбора отмечаемого пункта меню. (Флажки и поля со списками рассматриваются далее в книге.) Существуют два типа событий элементов, которые идентифицируются целочисленными константами, кратко описанными в табл. 25.4.

Таблица 25.4. Константы, определенные в классе `ItemEvent`

Константа	Описание
DESELECTED	Пользователь отменил выбор элемента
SELECTED	Пользователь выбрал элемент

Кроме того, в классе `ItemEvent` определена целочисленная константа `ITEM_STATE_CHANGED`, которая обозначает изменение состояния.

Вот конструктор класса `ItemEvent`:

```
ItemEvent(ItemSelectable src, int type, Object entry, int state)
```

В `src` передается ссылка на компонент, сгенерировавший событие. Например, им может быть список или выбранный элемент. Тип события указывается в `type`. Конкретный элемент, сгенерировавший событие элемента, передается в `entry`. Текущее состояние этого элемента указывается в `state`.

Метод `getItem()` можно использовать для получения ссылки на измененный элемент. Его сигнатура показана ниже:

```
Object getItem()
```

Метод `getItemSelectable()` можно применять для получения ссылки на объект `ItemSelectable`, сгенерировавший событие:

```
ItemSelectable getItemSelectable()
```

Списки и переключатели являются примерами элементов пользовательского интерфейса, которые реализуют интерфейс ItemSelectable.

Метод getStateChange() возвращает изменение состояния (т.е. SELECTED или DESELECTED) для события:

```
int getStateChange()
```

Класс KeyEvent

Событие типа KeyEvent генерируется, когда происходит ввод с клавиатуры. Существуют три типа ключевых событий, которые идентифицируются целочисленными константами KEY_PRESSED, KEY_RELEASED и KEY_TYPED. События первых двух типов возникают при нажатии или отпускании любой клавиши, а события третьего типа — только в случае появления символа. Помните, что не все нажатия клавиш дают в результате символ. Скажем, нажатие <Shift> не производит символ.

В классе KeyEvent определено множество других целочисленных констант. Например, константы VK_0–VK_9 и VK_A–VK_Z определяют ASCII-эквиваленты цифр и букв. Ниже перечислены остальные константы:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

Константы VK_X задают *коды виртуальных клавиш* и не зависят от модифицирующих клавиш, подобных <Control>, <Shift> или <Alt>. Класс KeyEvent является подклассом InputEvent. Вот один из его конструкторов:

```
KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)
```

В src передается ссылка на компонент, сгенерировавший событие. Тип события указывается в type, а системное время, когда была нажата клавиша, задается в when. Параметр modifiers отражает, какие модифицирующие клавиши были нажаты при возникновении события клавиатуры. Код виртуальной клавиши, такой как VK_UP, VK_A и т.д., указывается в code, а его символьный эквивалент (в случае существования) — в ch. Если допустимый символ не существует, то ch будет содержать CHAR_UNDEFINED. Для событий KEY_TYPED в code указывается VK_UNDEFINED.

В классе KeyEvent определено несколько методов, но наиболее часто используемые из них — getKeyChar(), возвращающий введенный символ, и getKeyCode(), который возвращает код клавиши.

Вот их общие формы:

```
char getKeyChar()
int getKeyCode()
```

В случае недоступности допустимого символа метод getKeyChar() возвращает CHAR_UNDEFINED. Когда возникает событие KEY_TYPED, метод getKeyCode() возвращает VK_UNDEFINED.

Класс MouseEvent

Есть восемь типов событий мыши. В классе MouseEvent определены целочисленные константы, кратко описанные в табл. 25.5, которые можно применять для их идентификации.

Таблица 25.5. Константы, определенные в классе MouseEvent

Константа	Описание
MOUSE_CLICKED	Пользователь щелкнул кнопкой мыши
MOUSE_DRAGGED	Пользователь перетащил указатель мыши
MOUSE_ENTERED	Указатель мыши наведен на компонент
MOUSE_EXITED	Указатель мыши покинул компонент
MOUSE_MOVED	Указатель мыши перемещен
MOUSE_PRESSED	Кнопка мыши нажата
MOUSE_RELEASED	Кнопка мыши отпущена
MOUSE_WHEEL	Колесико мыши было прокручено

Класс MouseEvent является подклассом InputEvent. Ниже представлен один из его конструкторов:

```
MouseEvent(Component src, int type, long when, int modifiers,
           int x, int y, int clicks, boolean triggersPopup)
```

В src передается ссылка на компонент, сгенерировавший событие. Тип события указывается в type, а системное время, когда произошло событие мыши, задается в when. Параметр modifiers определяет, какие модифицирующие клавиши были нажаты, когда произошло событие мыши. Координаты мыши указываются в x и y. Количество щелчков передается в clicks. Флаг triggersPopup определяет, приводит ли событие к появлению всплывающего меню на этой платформе.

Чаще всего в классе MouseEvent используются методы getX() и getY(). Они возвращают координаты X и Y указателя мыши внутри компонента, когда произошло событие. Вот их общие формы:

```
int getX()
int getY()
```

В качестве альтернативы для получения координат указателя мыши можно применять метод getPoint():

```
Point getPoint()
```

Он возвращает объект Point, который содержит координаты X и Y в своих целочисленных членах x и y.

Метод translatePoint() изменяет местоположение события:

```
void translatePoint(int x, int y)
```

Он добавляет к координатам события значения, переданные в x и y.

Метод getClickCount() получает количество щелчков кнопкой мыши для данного события. Вот его сигнатура:

```
int getClickCount()
```

Метод isPopupTrigger() проверяет, приводит ли событие к появлению всплывающего меню на текущей платформе. Он имеет следующую форму:

```
boolean isPopupTrigger()
```

Кроме того, доступен метод getButton():

```
int getButton()
```

Он возвращает значение, представляющее кнопку, которая стала причиной генерации события. В большинстве случаев возвращаемое значение будет одной из перечисленных ниже констант, определенных в MouseEvent:

NOBUTTON	BUTTON1	BUTTON2	BUTTON3

Значение NOBUTTON указывает на то, что ни одна из кнопок не была нажата или отпущена.

Доступны также три метода, которые получают координаты мыши относительно экрана, а не компонента:

```
Point getLocationOnScreen()
int getXOnScreen()
int getYOnScreen()
```

Метод getLocationOnScreen() возвращает объект Point, содержащий координаты X и Y. Два других метода возвращают координаты, отраженные в их именах.

Класс MouseWheelEvent

Класс MouseWheelEvent инкапсулирует событие колесика мыши и является подклассом MouseEvent. Колесико есть не у всех мышей. Если оно имеется, то обычно располагается между левой и правой кнопками. Колесико мыши используется для прокрутки. В классе MouseWheelEvent определены две целочисленные константы, кратко описанные в табл. 25.6.

Таблица 25.6. Константы, определенные в классе MouseWheelEvent

Константа	Описание
WHEEL_BLOCK_SCROLL	Произошло событие прокрутки на страницу вверх или вниз
WHEEL_UNIT_SCROLL	Произошло событие прокрутки на строку вверх или вниз

Ниже показан один из конструкторов, определенных в MouseWheelEvent:

```
MouseWheelEvent(Component src, int type, long when, int modifiers,
                int x, int y, int clicks, boolean triggersPopup,
                int scrollHow, int amount, int count)
```

В src передается ссылка на объект, сгенерировавший событие. Тип события указывается в type, а системное время, когда произошло событие мыши, задается в when. Параметр modifiers отражает, какие модифицирующие клавиши были нажаты при возникновении события. Координаты мыши передаются в x и y, а количество щелчков — в clicks. Флаг triggersPopup указывает, приводит ли событие к появлению всплывающего меню на этой платформе. Значением scrollHow должно быть либо WHEEL_UNIT_SCROLL, либо WHEEL_BLOCK_SCROLL. Количество единиц для прокрутки задается в amount. В count указывается количество единиц вращения, на которое было прокручено колесико. В классе MouseWheelEvent определены методы, предоставляющие доступ к событию колесика. Для получения количества единиц вращения понадобится вызвать метод getWheelRotation():

```
int getWheelRotation()
```

Он возвращает количество единиц вращения. Если значение положительное, то колесико прокручивалось против часовой стрелки, а если отрицательное — тогда по часовой стрелке. Кроме того, доступен метод getPreciseWheelRotation(), который поддерживает колесико с высоким разрешением. Он работает подобно getWheelRotation(), но возвращает значение типа double.

Для получения типа прокрутки необходимо вызвать метод getScrollType():

```
int getScrollType()
```

Он возвращает либо WHEEL_UNIT_SCROLL, либо WHEEL_BLOCK_SCROLL.

В случае типа прокрутки WHEEL_UNIT_SCROLL посредством метода getScrollAmount() можно получить количество единиц прокрутки:

```
int getScrollAmount()
```

Класс TextEvent

Экземпляры класса TextEvent описывают события, связанные с текстом, которые генерируются текстовыми полями и областями, когда символы вводятся пользователем или программой. В классе TextEvent определена целочисленная константа TEXT_VALUE_CHANGED.

Ниже показан один конструктор данного класса:

```
TextEvent(Object src, int type)
```

В src передается ссылка на объект, сгенерировавший событие. Тип события указывается в type.

Объект TextEvent не включает символы, в текущий момент находящиеся в текстовом компоненте, который сгенерировал событие. Взамен для получения

такой информации программа должна применять другие методы, связанные с текстовым компонентом. Эта операция отличается от других объектов событий, обсуждаемых в настоящем разделе. Считайте уведомление о событии, которое связано с текстом, своего рода сигналом прослушивателю о том, что он должен получить информацию из определенного текстового компонента.

Класс WindowEvent

Существуют десять типов оконных событий. В классе WindowEvent определены целочисленные константы, которые можно использовать для их идентификации. Константы кратко описаны в табл. 25.7.

Таблица 25.7. Константы, определенные в классе WindowEvent

Константа	Описание
WINDOW_ACTIVATED	Окно стало активным
WINDOW_CLOSED	Окно было закрыто
WINDOW_CLOSING	Пользователь запросил закрытие окна
WINDOW_DEACTIVATED	Окно перестало быть активным
WINDOW_DEICONIFIED	Окно было развернуто
WINDOW_GAINED_FOCUS	Окно получило фокус ввода
WINDOW_ICONIFIED	Окно было свернуто
WINDOW_LOST_FOCUS	Окно утратило фокус ввода
WINDOW_OPENED	Окно было открыто
WINDOW_STATE_CHANGED	Состояние окна изменилось

Класс WindowEvent является подклассом ComponentEvent. В нем определено несколько конструкторов. Вот первый:

```
WindowEvent(Window src, int type)
```

В src передается ссылка на объект, который сгенерировал событие типа, указанного в type.

Следующие три конструктора предлагают более точный контроль над созданием событий:

```
WindowEvent(Window src, int type, Window other)
WindowEvent(Window src, int type, int fromState, int toState)
WindowEvent(Window src, int type, Window other, int fromState, int toState)
```

В other задается противоположное окно, когда происходит событие, связанное с фокусом или активностью окна. В fromState указывается предыдущее состояние окна, а в toState — новое состояние, в которое перейдет окно при смене состояния. Чаще всего применяется метод getWindow() класса WindowEvent. Он возвращает объект Window, сгенерировавший событие.

Ниже показан его общий вид:

```
Window getWindow()
```

В классе `WindowEvent` также определены методы, которые возвращают противоположное окно (при возникновении события, связанного с фокусом или активностью окна), предыдущее состояние окна и текущее состояние окна:

```
Window getOppositeWindow()
int getOldState()
int getNewState()
```

Источники событий

В табл. 25.8 кратко описаны компоненты пользовательского интерфейса, которые могут генерировать события, описанные в предыдущем разделе. В дополнение к таким элементам графического пользовательского интерфейса генерировать события способен любой класс, производный от `Component`, скажем, `Frame`. Например, от экземпляра `Frame` можно получать события клавиатуры и мыши. В этой главе обрабатываются только события мыши и клавиатуры, но в последующих главах будут обрабатываться события из ряда других источников.

Таблица 25.8. Примеры источников событий

Источник событий	Описание
Кнопка	Генерирует события действий, когда кнопка нажата
Флажок	Генерирует события элементов, когда выбирается флажок или его выбор отменяется
Переключатель	Генерирует события элементов при изменении переключателя
Список	Генерирует события действий при двойном щелчке на элементе и события элементов, когда элемент выбирается или его выбор отменяется
Пункт меню	Генерирует события действий при выборе пункта меню и события элементов, когда отмечаемый пункт меню выбирается или его выбор отменяется
Полоса прокрутки	Генерирует события корректировки при манипулировании полосой прокрутки
Текстовые компоненты	Генерирует события текста, когда пользователь вводит символы
Окно	Генерирует события окна, когда окно становится активным, закрывается, перестает быть активным, разворачивается, сворачивается, открывается или закрывается

Интерфейсы прослушивателей событий

Как объяснялось ранее, модель делегирования обработки событий состоит из двух частей: источников и прослушивателей. Что касается настоящей главы, то прослушиватели создаются за счет реализации одного или нескольких интерфейсов, определенных в пакете `java.awt.event`. Когда происходит событие, источник события вызывает надлежащий метод, который определен в классе прослушивателя, и предоставляет объект события в качестве аргумента. В табл. 25.9 перечислено несколько часто используемых интерфейсов прослушивателей и приведено краткое описание методов, которые они определяют. В последующих разделах рассматриваются конкретные методы, содержащиеся в каждом интерфейсе.

Таблица 25.9. Часто используемые интерфейсы прослушивателей событий

Интерфейс	Описание
`ActionListener`	Определяет один метод для получения событий действий
`AdjustmentListener`	Определяет один метод для получения событий корректировки
`ComponentListener`	Определяет четыре метода для распознавания ситуаций, когда компонент скрывается, перемещается, изменяет размер или показывается
`ContainerListener`	Определяет два метода для распознавания ситуаций, когда компонент добавляется в контейнер или удаляется из него
`FocusListener`	Определяет два метода для распознавания ситуаций, когда компонент получает или утрачивает фокус
`ItemListener`	Определяет один метод для распознавания ситуации, когда состояние элемента изменяется
`KeyListener`	Определяет три метода для распознавания ситуаций, когда клавиша нажимается, отпускается или с ее помощью вводится символ
`MouseListener`	Определяет пять методов для распознавания ситуаций, когда указатель мыши наводится на компонент или покидает его, совершается щелчок кнопкой мыши, кнопка мыши нажимается или отпускается
`MouseMotionListener`	Определяет два метода для распознавания ситуаций, когда указатель мыши перетаскивается или перемещается

Окончание табл. 25.9

Интерфейс	Описание
MouseWheelListener	Определяет один метод для распознавания ситуации, когда колесико мыши прокручивается
TextListener	Определяет один метод для распознавания ситуации, когда текстовое значение изменяется
WindowFocusListener	Определяет два метода для распознавания ситуаций, когда окно получает или утрачивает фокус ввода
WindowListener	Определяет семь методов для распознавания ситуаций, когда окно становится активным, закрывается, перестает быть активным, разворачивается, сворачивается, открывается или закрывается

Интерфейс ActionListener

В интерфейсе ActionListener определен метод actionPerformed(), который вызывается при возникновении события действия. Вот его общая форма:

```
void actionPerformed(ActionEvent ae)
```

Интерфейс AdjustmentListener

В интерфейсе AdjustmentListener определен метод adjustmentValue Changed(), который вызывается при возникновении события корректировки. Ниже показана его общая форма:

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

Интерфейс ComponentListener

В интерфейсе ComponentListener определены четыре метода, которые вызываются, когда компонент изменяет размер, перемещается, показывается или скрывается. Их общие формы выглядят следующим образом:

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

Интерфейс ContainerListener

В интерфейсе ContainerListener определены два метода. Когда компонент добавляется в контейнер, вызывается метод componentAdded(), а при удалении компонента из контейнера вызывается componentRemoved(). Вот их общие формы:

```
void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)
```

Интерфейс FocusListener

В интерфейсе FocusListener определены два метода. Когда компонент получает фокус ввода с клавиатуры, вызывается метод focusGained(), а когда компонент утрачивает фокус — метод focusLost(). Ниже приведены их общие формы:

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```

Интерфейс ItemListener

В интерфейсе ItemListener определен метод itemStateChanged(), вызываемый при изменении состояния элемента, который имеет следующую общую форму:

```
void itemStateChanged(ItemEvent ie)
```

Интерфейс KeyListener

В интерфейсе KeyListener определены три метода. Методы keyPressed() и keyReleased() вызываются соответственно при нажатии и отпускании клавиши. Метод keyTyped() вызывается в случае, если был введен символ.

Например, если пользователь нажимает и отпускает клавишу <A>, тогда последовательно генерируются три события: нажатие клавиши, ввод символа и отпускание клавиши. Если пользователь нажимает и отпускает клавишу <Home>, то последовательно генерируются два ключевых события: нажатие клавиши и отпускание клавиши.

Далее представлены общие формы упомянутых методов:

```
void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)
```

Интерфейс MouseListener

В интерфейсе MouseListener определены пять методов. Если кнопка мыши нажата и отпущена в одной и той же точке, тогда вызывается метод mouseClicked(). Когда указатель мышь наводится на компонент, вызывается метод mouseEntered(), а когда он покидает компонент, вызывается mouseExited(). Методы mousePressed() и mouseReleased() вызываются при нажатии и отпускании кнопки мыши.

Вот общие формы этих методов:

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

Интерфейс MouseMotionListener

В интерфейсе MouseMotionListener определены два метода. Метод mouseDragged() вызывается множество раз во время перетаскивания указателя мыши, а метод mouseMoved() — множество раз при перемещении мыши. Ниже показаны их общие формы:

```
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

Интерфейс MouseWheelListener

В интерфейсе MouseWheelListener определен метод mouseWheelMoved(), который вызывается при прокручивании колесика мыши:

```
void  mouseWheelMoved(MouseWheelEvent  mwe)
```

Интерфейс TextListener

В интерфейсе TextListener определен метод textValueChanged(), который вызывается при внесении изменения в текстовую область или в текстовое поле:

```
void textValueChanged(TextEvent te)
```

Интерфейс WindowFocusListener

В интерфейсе WindowFocusListener определены два метода: windowGained Focus() и windowLostFocus(). Они вызываются, когда окно получает или теряет фокус ввода. Вот их общие формы:

```
void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
```

Интерфейс WindowListener

В интерфейсе WindowListener определены семь методов. Методы windowActivated() и windowDeactivated() вызываются, когда окно становится или перестает быть активным. Если окно сворачивается, тогда вызывается метод windowIconified(), а если разворачивается — то метод windowDeiconified(). Когда окно открывается или закрывается, вызывается метод windowOpened() или windowClosed(). Метод windowClosing() вызывается при закрытии окна. Общие формы упомянутых методов выглядят следующим образом:

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

Использование модели делегирования обработки событий

Теперь, когда вы изучили теорию, лежащую в основе модели события делегирования, и ознакомились с ее компонентами, наступило время увидеть ее на практике. Использовать модель делегирования обработки событий на самом деле довольно легко. Понадобится просто выполнить два описанных ниже шага.

1. Реализовать подходящий интерфейс в прослушивателе, чтобы он мог принимать события желаемого типа.
2. Реализовать код для регистрации и отмены регистрации (при необходимости) прослушивателя как получателя уведомлений о событиях.

Не забывайте, что источник может генерировать события нескольких типов. Каждый тип событий должен регистрироваться отдельно. Кроме того, объект может быть зарегистрирован для получения нескольких типов событий, но тогда он должен реализовать все интерфейсы, необходимые для получения этих событий. Во всех случаях обработчик событий обязан быстро возвращать управление. Как объяснялось ранее, обработчик событий не должен удерживать управление в течение длительного периода времени.

Чтобы взглянуть, каким образом модель делегирования работает на практике, рассмотрим примеры, имеющие дело с двумя распространенными генераторами событий, которыми является мышь и клавиатура.

Основные концепции графических пользовательских интерфейсов AWT

Для демонстрации основ обработки событий будут применяться простые программы с графическим пользовательским интерфейсом. Как утверждалось ранее, большинство событий, на которые реагирует программа, генерируется благодаря взаимодействию пользователя с программами, имеющими графический пользовательский интерфейс. Хотя программы с графическим пользовательским интерфейсом, представленные в настоящей главе, очень просты, все же необходимо прояснить несколько основных концепций, поскольку такие программы отличаются от консольных программ, которые встречаются в других местах книги.

Прежде всего, важно отметить, что все современные версии Java поддерживают две инфраструктуры для построения графических пользовательских интерфейсов: AWT и Swing. Библиотека AWT была первой инфраструктурой, которую проще всего использовать для крайне ограниченных программ с графическим пользовательским интерфейсом. Вторая инфраструктура, Swing, является наиболее популярной и широко применяемой. (С несколькими недавними версиями Java поставлялась третья инфраструктура для создания графических пользовательских интерфейсов, называемая JavaFX. Тем не

менее, начиная с JDK 11, она больше не входит в состав JDK.) Далее в книге обсуждается как AWT, так и Swing. Однако для демонстрации фундаментальных основ обработки событий вполне достаточно простых программ с графическим пользовательским интерфейсом на основе AWT.

В последующих программах используются четыре ключевых функциональных возможности AWT. Во-первых, во всех программах создаются окна верхнего уровня за счет расширения класса `Frame`, который определяет так называемое "нормальное" окно. Скажем, у него есть кнопки для сворачивания, разворачивания и закрытия. Можно изменять его размеры, скрывать и отображать заново.

Во-вторых, во всех программах переопределяется метод `paint()`, отвечающий за вывод в окно, который вызывается исполняющей средой для отображения вывода в окне. Например, метод `paint()` вызывается при первом отображении окна и после того, как окно было скрыто, а затем снова показано. В-третьих, когда программе нужно отобразить вывод, метод `paint()` напрямую не вызывается. Взамен вызывается метод `repaint()`. По существу `repaint()` сообщает инфраструктуре AWT о необходимости вызова `paint()`. Работа этого процесса демонстрируется в примерах далее в главе. Наконец, в-четвертых, когда окно `Frame` верхнего уровня в приложении закрывается (скажем, в результате щелчка на кнопке закрытия), программа должна явным образом завершиться, часто через вызов `System.exit()`. Сам по себе щелчок на кнопке закрытия не приводит к завершению программы. Следовательно, программа с графическим пользовательским интерфейсом на основе AWT должна обрабатывать событие закрытия окна.

Обработка событий мыши

Для обработки событий мыши понадобится реализовать интерфейсы `MouseListener` и `MouseMotionListener`. (Можно также реализовать интерфейс `MouseWheelListener`, но здесь это не делается.) Процесс продемонстрирован в следующей программе, которая отображает текущие координаты мыши в своем окне. При каждом нажатии кнопки в месте нахождения указателя мыши отображается сообщение **Button Down** (Кнопка нажата), а при каждом отпускании кнопки — сообщение **Button Released** (Кнопка отпущена). Если произведен щелчок кнопкой мыши, тогда в текущем местоположении мыши отображается сообщение о данном факте.

Когда указатель мыши наводится на окно или покидает его, отображается сообщение **Mouse entered** (Указатель мыши наведен на окно) или **Mouse exited** (Указатель мыши покинул окно). При перетаскивании мыши отображается символ *, который следует за указателем мыши по мере его перетаскивания. Обратите внимание, что две переменные, `mouseX` и `mouseY`, хранят местоположение указателя мыши, когда происходят события нажатия кнопки, отпускания кнопки или перетаскивания указателя мыши. Затем координаты `mouseX` и `mouseY` задействуются в методе `paint()` для отображения вывода в точке, где возникали события.

```
// Демонстрация обработчиков событий мыши.
import java.awt.*;
import java.awt.event.*;
public class MouseEventsDemo
       extends Frame implements MouseListener, MouseMotionListener {

  String msg = "";
  int mouseX = 0, mouseY = 0;  // координаты указателя мыши

  public MouseEventsDemo() {
    addMouseListener(this);
    addMouseMotionListener(this);
    addWindowListener(new MyWindowAdapter());
  }

  // Обработать щелчок кнопкой мыши.
  public void mouseClicked(MouseEvent me) {
    msg = msg + " -- click received"; repaint();
              // -- получен щелчок
  }

  // Обработать наведение на окно указателя мыши.
  public void mouseEntered(MouseEvent me) {
    mouseX = 100;
    mouseY = 100;
    msg = "Mouse entered";
        // Указатель мыши наведен на окно
    repaint();
  }

  // Обработать покидание окна указателем мыши.
  public void mouseExited(MouseEvent me) {
    mouseX = 100;
    mouseY = 100;
    msg = "Mouse exited";
        // Указатель мыши покинул окно
    repaint();
  }

  // Обработать нажатие кнопки мыши.
  public void mousePressed(MouseEvent me) {
    // Сохранить координаты.
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Button down";
        // Кнопка нажата
    repaint();
  }

  // Обработать отпускание кнопки мыши.
  public void mouseReleased(MouseEvent me) {
    // Сохранить координаты.
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Button Released";
         // Кнопка отпущена
    repaint();
  }
```

```
  // Обработать перетаскивание указателя мыши.
  public void mouseDragged(MouseEvent me) {
    // Сохранить координаты.
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "*" + " mouse at " + mouseX + ", " + mouseY;
        // * курсор мыши находится в
    repaint();
  }

  // Обработать перемещение указателя мыши.
  public void mouseMoved(MouseEvent me) {
    msg = "Moving mouse at " + me.getX() + ", " + me.getY();
        // Перемещение курсора мыши в
    repaint();
  }

  // Отобразить сообщение в текущей позиции X, Y окна.
  public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
  }

  public static void main(String[] args) {
    MouseEventsDemo appwin = new MouseEventsDemo();
    appwin.setSize(new Dimension(300, 300));
    appwin.setTitle("MouseEventsDemo");
    appwin.setVisible(true);
  }
}

// При щелчке на кнопке закрытия закрыть окно и завершить программу.
class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

Пример вывода программы показан на рис. 25.1.

Рис. 25.1. Пример вывода программы, демонстрирующей обработку событий мыши

Давайте внимательно рассмотрим пример. Первым делом обратите внимание, что класс MouseEventsDemo расширяет Frame. Таким образом, он формирует окно верхнего уровня для приложения. Далее обратите внимание на реализацию им интерфейсов MouseListener и MouseMotionListener, которые содержат методы, получающие и обрабатывающие события мыши различных типов. Кроме того, класс MouseEventsDemo является одновременно источником и прослушивателем этих событий.

Дело в том, что Frame предоставляет методы addMouseListener() и addMouseMotionListener(). Применение одного и того же класса в качестве источника и прослушивателя событий — вовсе не редкость в простых программах с графическим пользовательским интерфейсом.

Внутри конструктора MouseEventsDemo программа регистрируется как прослушиватель событий мыши, для чего вызываются методы addMouseListener() и addMouseMotionListener():

```
void addMouseListener(MouseListener ml)
void addMouseMotionListener(MouseMotionListener mml)
```

В ml передается ссылка на объект, получающий события мыши, а в mml — ссылка на объект, получающий события перемещения мыши. В программе обе ссылки указывают на тот же самый объект. Затем класс MouseEventsDemo реализует все методы, определенные интерфейсами MouseListener и MouseMotionListener, т.е. обработчики разнообразных событий мыши. Каждый метод обрабатывает свое событие и затем возвращает управление.

Обратите внимание, что в конструкторе класса MouseEventsDemo также добавляется объект реализации WindowListener. Он необходим для того, чтобы программа реагировала на событие закрытия окна, когда пользователь щелкает на кнопке закрытия. Для реализации интерфейса WindowListener этот прослушиватель использует *класс адаптера*. Классы адаптеров предоставляют пустые реализации интерфейса прослушивателя, позволяя переопределять только интересующие методы. Они подробно описаны далее в главе, но здесь применяется один из них, чтобы значительно упростить код, необходимый для закрытия программы. В данном случае переопределяется метод windowClosing(), который инфраструктура AWT вызывает при закрытии окна. В нем производится вызов System.exit(), чтобы завершить программу.

Далее обратите внимание на обработчики событий мыши. Каждый раз, когда происходит событие мыши, msg присваивается строка, описывающая, что произошло, и затем вызывается метод repaint(). Здесь метод repaint() в конечном итоге заставляет AWT вызывать paint() для отображения вывода. (Более подробно процесс будет исследован в главе 26.) Как видите, метод paint() принимает параметр типа Graphics. Класс Graphics описывает *графический контекст* программы, который нужен для вывода. В программе используется метод drawString() класса Graphics для фактического отображения строки в указанном местоположении X, Y окна. Ниже показана форма метода drawString(), применяемая в программе:

```
void drawString(String message, int x, int y)
```

В message указывается строка, которая должна быть выведена, начиная с позиции x, y. В окне Java верхний левый угол находится в позиции 0, 0. Как уже упоминалось, переменные mouseX и mouseY отслеживают местоположение указателя мыши. Их значения передаются методу drawString() в качестве позиции для отображения вывода.

Наконец, программа запускается путем создания экземпляра MouseEvents Demo, после чего устанавливаются размеры окна и его заголовок, а затем окно делается видимым. Соответствующие функциональные средства обсуждаются в главе 26.

Обработка событий клавиатуры

Для обработки событий клавиатуры используется та же общая архитектура, что и в примере с событиями мыши в предыдущем разделе. Конечно, отличие в том, что будет реализован интерфейс KeyListener.

Прежде чем заняться примером, полезно выяснить, как генерируются основные события. При нажатии клавиши генерируется событие KEY_PRESSED, которое приводит к вызову обработчика события keyPressed(). При отпускании клавиши генерируется событие KEY_RELEASED и запускается обработчик keyReleased(). Если в результате нажатия клавиши вводится символ, тогда отправляется событие KEY_TYPED и вызывается обработчик keyTyped(). Таким образом, каждый раз, когда пользователь нажимает клавишу, генерируются как минимум два, а зачастую и три события. Если вас интересуют лишь реальные символы, то можете игнорировать информацию, передаваемую событиями нажатия и отпускания клавиши. Тем не менее, если программа должна обрабатывать специальные клавиши вроде клавиш со стрелками или функциональных клавиш, тогда придется отслеживать их с помощью обработчика keyPressed().

В показанной далее программе демонстрируется ввод с клавиатуры. В окне отображаются результаты нажатых клавиш вместе с их состоянием.

```
// Демонстрация обработчиков событий клавиатуры.

import java.awt.*;
import java.awt.event.*;

public class SimpleKey extends Frame implements KeyListener {

  String msg = "";
  String keyState = "";

  public SimpleKey() {
    addKeyListener(this);
    addWindowListener(new MyWindowAdapter());
  }

  // Обработать нажатие клавиши.
  public void keyPressed(KeyEvent ke) {
    keyState = "Key Down";         // клавиша нажата
    repaint();
  }
```

```
  // Обработать отпускание клавиши.
  public void keyReleased(KeyEvent ke) {
    keyState = "Key Up";          // клавиша отпущена
    repaint();
  }

  // Обработать ввод символа.
  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
  }

  // Отобразить результаты нажатых клавиш.
  public void paint(Graphics g) {
    g.drawString(msg, 20, 100);
    g.drawString(keyState, 20, 50);
  }

  public static void main(String[] args) {
    SimpleKey appwin = new SimpleKey();

    appwin.setSize(new Dimension(200, 150));
    appwin.setTitle("SimpleKey");
    appwin.setVisible(true);
  }
}

// При щелчке на кнопке закрытия закрыть окно и завершить программу.
class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

Пример вывода программы показан на рис. 25.2.

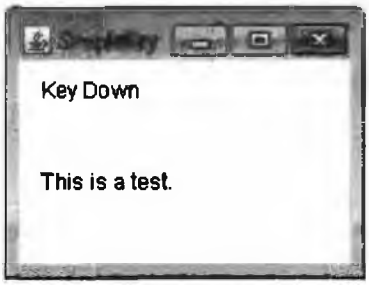


Рис. 25.2. Пример вывода программы, демонстрирующей обработку событий клавиатуры

Если нужно обрабатывать специальные клавиши вроде клавиш со стрелками или функциональных клавиш, тогда реагировать на них придется в обработчике `keyPressed()`. В `keyTyped()` они недоступны. Для идентификации клавиш применяются коды виртуальных клавиш.

Скажем, приведенная ниже программа выводит названия нескольких специальных клавиш:

```
// Демонстрация нескольких кодов виртуальных клавиш.
import java.awt.*;
import java.awt.event.*;
public class KeyEventsDemo extends Frame implements KeyListener {
  String msg = "";
  String keyState = "";
  public KeyEventsDemo() {
    addKeyListener(this);
    addWindowListener(new MyWindowAdapter());
  }
  // Обработать нажатие клавиши.
  public void keyPressed(KeyEvent ke) {
    keyState = "Key Down";         // клавиша нажата
    int key = ke.getKeyCode();
    switch(key) {
      case KeyEvent.VK_F1:
        msg += "<F1>";
        break;
      case KeyEvent.VK_F2:
        msg += "<F2>";
        break;
      case KeyEvent.VK_F3:
        msg += "<F3>";
        break;
      case KeyEvent.VK_PAGE_DOWN:
        msg += "<PgDn>";
        break;
      case KeyEvent.VK_PAGE_UP:
        msg += "<PgUp>";
        break;
      case KeyEvent.VK_LEFT:
        msg += "<Left Arrow>";     // стрелка влево
        break;
      case KeyEvent.VK_RIGHT:
        msg += "<Right Arrow>";    // стрелка вправо
        break;
    }
    repaint();
  }
  // Обработать отпускание клавиши.
  public void keyReleased(KeyEvent ke) {
    keyState = "Key Up";           // клавиша отпущена
    repaint();
  }
  // Обработать ввод символа.
  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
  }
  // Отобразить результаты нажатых клавиш.
  public void paint(Graphics g) {
    g.drawString(msg, 20, 100);
    g.drawString(keyState, 20, 50);
  }
```

```
  public static void main(String[] args) {
    KeyEventsDemo appwin = new KeyEventsDemo();
    appwin.setSize(new Dimension(200, 150));
    appwin.setTitle("KeyEventsDemo");
    appwin.setVisible(true);
  }
}
// При щелчке на кнопке закрытия закрыть окно и завершить программу.
class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

Пример вывода программы представлен на рис. 25.3.

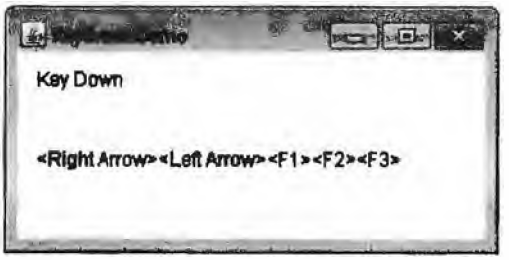


Рис. 25.3. Пример вывода программы, демонстрирующей обработку специальных клавиш

Процедуры из предыдущих примеров обработки событий клавиатуры и мыши можно обобщить для любого вида обработки событий, включая события, которые генерируются элементами управления. В последующих главах вы встретите много примеров обработки других типов событий, но все они будут иметь такую же базовую структуру, как в только что описанных программах.

Классы адаптеров

В Java есть специальное средство под названием *класс адаптера*, которое может упростить создание обработчиков событий в определенных ситуациях. Класс адаптера предоставляет пустые реализации всех методов из интерфейса прослушивателя событий. Классы адаптеров полезны, когда желательно получать и обрабатывать лишь некоторые события, поддерживаемые конкретным интерфейсом прослушивателя событий. Для определения нового класса для прослушивания событий можно расширить один из классов адаптеров и реализовать только интересующие методы.

Например, класс `MouseMotionAdapter` имеет два метода, `mouseDragged()` и `mouseMoved()`, которые определены в интерфейсе `MouseMotionListener`.

Если нужны только события перетаскивания указателя мыши, тогда можно просто расширить MouseMotionAdapter и переопределить метод mouseDragged(), а события перемещения указателя мыши обработает пустая реализация mouseMoved().

В табл. 25.10 кратко описаны часто используемые классов адаптеров из java.awt.event вместе с интерфейсами прослушивателей, которые они реализуют.

Таблица 25.10. Часто используемые классы адаптеров, реализующие интерфейсы прослушивателей

Класс	Интерфейс прослушивателя
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener, MouseMotionListener и MouseWheelListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener, WindowFocusListener и WindowStateListener

В предшествующих примерах вы уже видели один класс адаптера в действии — WindowAdapter. Вспомните, что в интерфейсе WindowListener определены семь методов, но в программах был нужен только метод windowClosing(). Применение адаптера избавило от необходимости создавать пустые реализации для остальных неиспользуемых методов, позволив избежать беспорядка в примерах. Как и следовало ожидать, другие классы адаптеров можно применять аналогичным образом.

В следующей программе демонстрируется еще один пример использования класса адаптера MouseAdapter для реагирования на события щелчков кнопкой мыши и перетаскивания ее указателя. Как показано в табл. 25.10, класс MouseAdapter реализует все интерфейсы прослушивателя событий мыши, что позволяет его применять для обработки всех типов событий мыши. Разумеется, необходимо переопределять только те методы, которые задействованы в программе. Класс MyMouseAdapter расширяет MouseAdapter и переопределяет методы mouseClicked() и mouseDragged(). Все остальные события мыши игнорируются. Обратите внимание, что конструктору MyMouseAdapter передается ссылка на экземпляр AdapterDemo, которая сохраняется и затем используется для присваивания переменной msg строки и вызова метода repaint() на объекте, получающем уведомление о событии. Как и прежде, класс WindowAdapter применяется для обработки события закрытия окна.

```
// Демонстрация использования классов адаптеров.
import java.awt.*;
import java.awt.event.*;

public class AdapterDemo extends Frame {
  String msg = "";

  public AdapterDemo() {
    addMouseListener(new MyMouseAdapter(this));
    addMouseMotionListener(new MyMouseAdapter(this));
    addWindowListener(new MyWindowAdapter());
  }

  // Отобразить информацию, связанную с мышью.
  public void paint(Graphics g) {
    g.drawString(msg, 20, 80);
  }

  public static void main(String[] args) {
    AdapterDemo appwin = new AdapterDemo();

    appwin.setSize(new Dimension(200, 150));
    appwin.setTitle("AdapterDemo");
    appwin.setVisible(true);
  }
}

// Обрабатывать только события щелчка и перетаскивания.
class MyMouseAdapter extends MouseAdapter {
  AdapterDemo adapterDemo;

  public MyMouseAdapter(AdapterDemo adapterDemo) {
    this.adapterDemo = adapterDemo;
  }

  // Обработать щелчок кнопкой мыши.
  public void mouseClicked(MouseEvent me) {
    adapterDemo.msg = "Mouse clicked";
                      // Щелчок кнопкой мыши
    adapterDemo.repaint();
  }

  // Обработать перетаскивание указателя мыши.
  public void mouseDragged(MouseEvent me) {
    adapterDemo.msg = "Mouse dragged";
                      // Перетаскивание указателя мыши
    adapterDemo.repaint();
  }
}

// При щелчке на кнопке закрытия закрыть окно и завершить программу.
class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

Как видно в программе, отсутствие необходимости реализовывать все методы, определенные интерфейсами MouseMotionListener, MouseListener и MouseWheelListener, экономит значительные усилия и предотвращает загромождение кода пустыми методами. В качестве упражнения можете попробовать переписать один из показанных ранее примеров с вводом с клавиатуры, чтобы задействовать в нем класс KeyAdapter.

Внутренние классы

В главе 7 объяснялись основы внутренних классов, а здесь вы узнаете, почему они важны. Как вам уже известно, *внутренний класс* — это класс, определенный в другом классе или даже в выражении. В настоящем разделе показано, как с помощью внутренних классов можно упростить код при использовании классов адаптеров событий.

Чтобы понять преимущества внутренних классов, взгляните на приведенную далее программу. Внутренний класс в ней *не* применяется. Цель программы — отображение строки Mouse Pressed (Кнопка мыши нажата) в случае нажатия кнопки мыши. Подобно подходу, использованному в предыдущем примере, конструктору MyMouseAdapter передается ссылка на экземпляр MousePressedDemo, которая затем сохраняется и применяется для присваивания переменной msg строки и вызова метода repaint() на объекте, получающем уведомление о событии.

```
// В этой программе внутренний класс НЕ используется.
import java.awt.*;
import java.awt.event.*;

public class MousePressedDemo extends Frame {
  String msg = "";

  public MousePressedDemo() {
    addMouseListener(new MyMouseAdapter(this));
    addWindowListener(new MyWindowAdapter());
  }
  public void paint(Graphics g) {
    g.drawString(msg, 20, 100);
  }

  public static void main(String[] args) {
    MousePressedDemo appwin = new MousePressedDemo();

    appwin.setSize(new Dimension(200, 150));
    appwin.setTitle("MousePressedDemo");
    appwin.setVisible(true);
  }
}

class MyMouseAdapter extends MouseAdapter {
  MousePressedDemo mousePressedDemo;

  public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
    this.mousePressedDemo = mousePressedDemo;
  }
```

```
  // Обработать нажатие кнопки мыши.
  public void mousePressed(MouseEvent me) {
    mousePressedDemo.msg = "Mouse Pressed";
                          // Кнопка мыши нажата
    mousePressedDemo.repaint();
  }
}
// При щелчке на кнопке закрытия закрыть окно и завершить программу.
class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

Ниже показано, как можно улучшить предыдущую программу за счет использования внутреннего класса. Здесь `InnerClassDemo` — класс верхнего уровня, а `MyMouseAdapter` — внутренний класс. Поскольку `MyMouseAdapter` определен в области видимости `InnerClassDemo`, он имеет доступ ко всем переменным и методам данного класса. Следовательно, метод `mousePressed()` способен напрямую вызывать метод `repaint()`. Делать это через сохраненную ссылку больше не нужно. То же самое касается и присваивания значения переменной `msg`. Таким образом, нет необходимости передавать `MyMouseAdapter()` ссылку на вызывающий объект. Также обратите внимание, что `MyWindowAdapter` был превращен во внутренний класс.

```
// Демонстрация использования внутренних классов.

import java.awt.*;
import java.awt.event.*;

public class InnerClassDemo extends Frame {
  String msg = "";

  public InnerClassDemo() {
    addMouseListener(new MyMouseAdapter());
    addWindowListener(new MyWindowAdapter());
  }
  // Внутренний класс для обработки событий нажатия кнопки мыши.
  class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent me) {
      msg = "Mouse Pressed";
          // Кнопка мыши нажата
      repaint();
    }
  }
  // Внутренний класс для обработки событий закрытия окна.
  class MyWindowAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent we) {
      System.exit(0);
    }
  }
  public void paint(Graphics g) {
    g.drawString(msg, 20, 80);
  }
```

```
  public static void main(String[] args) {
    InnerClassDemo appwin = new InnerClassDemo();

    appwin.setSize(new Dimension(200, 150));
    appwin.setTitle("InnerClassDemo");
    appwin.setVisible(true);
  }
}
```

Анонимные внутренние классы

Анонимный внутренний класс представляет собой класс, которому не назначено имя. В этом разделе показано, каким образом анонимный внутренний класс может облегчить написание обработчиков событий. Взгляните на показанную ниже программу, целью которой, как и прежде, является отображение строки Mouse Pressed (Кнопка мыши нажата) при нажатии кнопки мыши.

```
// Демонстрация использования анонимных внутренних классов.

import java.awt.*;
import java.awt.event.*;

public class AnonymousInnerClassDemo extends Frame {
  String msg = "";

  public AnonymousInnerClassDemo() {

    // Анонимный внутренний класс для обработки событий нажатия кнопки мыши.
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        msg = "Mouse Pressed";
            // Кнопка мыши нажата
        repaint();
      }
    });

    // Анонимный внутренний класс для обработки событий закрытия окна.
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    g.drawString(msg, 20, 80);
  }

  public static void main(String[] args) {
    AnonymousInnerClassDemo appwin =
      new AnonymousInnerClassDemo();

    appwin.setSize(new Dimension(200, 150));
    appwin.setTitle("AnonymousInnerClassDemo");
    appwin.setVisible(true);
  }
}
```

В программе есть один класс верхнего уровня — AnonymousInnerClassDemo. Его конструктор вызывает метод addMouseListener(), которому в качестве аргумента передается выражение, определяющее и создающее экземпляр анонимного внутреннего класса. Давайте внимательно проанализируем такое выражение.

Синтаксис new MouseAdapter() {...} указывает компилятору на то, что код между фигурными скобками определяет анонимный внутренний класс, который вдобавок расширяет класс MouseAdapter. Новый класс не имеет имени, но он автоматически создается при выполнении данного выражения. Приведенный синтаксис можно обобщить и он будет таким же при создании других анонимных классов, например, когда в программе создается анонимный WindowAdapter.

Так как анонимный внутренний класс определен в области видимости AnonymousInnerClassDemo, он имеет доступ ко всем переменным и методам этого класса. Следовательно, он может напрямую вызывать метод repaint() и обращаться к переменной msg.

Как было проиллюстрировано выше, именованные и анонимные внутренние классы решают некоторые надоедливые задачи простым, но эффективным способом. Они также позволяют получать более эффективный код.

ГЛАВА

26 Введение в AWT: работа с окнами, графикой и текстом

Библиотека AWT (Abstract Window Toolkit) была первой инфраструктурой для построения графических пользовательских интерфейсов Java, начиная с версии 1.0. Она содержит множество классов и методов, позволяющих создавать окна и простые элементы управления. Библиотека AWT была представлена в главе 25, где использовалась в нескольких коротких примерах, демонстрирующих обработку событий. В этой главе начинается более ее подробное исследование. Здесь вы узнаете, как управлять окнами, работать со шрифтами, выводить текст и задействовать графику. В главе 27 рассматриваются различные элементы управления AWT, диспетчеры компоновки и меню. В ней также объясняются дополнительные аспекты механизма обработки событий Java. В главе 28 предлагается введение в подсистему AWT, отвечающую за работу с изображениями.

Первым делом важно отметить, что вам редко придется создавать графические пользовательские интерфейсы, основанные исключительно на AWT, поскольку для таких целей были разработаны более мощные инфраструктуры Java (вроде Swing). Тем не менее, библиотека AWT остается важной частью Java. Давайте выясним почему.

На момент написания книги самой широко применяемой инфраструктурой была Swing. Так как Swing предлагает более развитый и гибкий механизм построения графических пользовательских интерфейсов по сравнению с AWT, легко сделать вывод, что библиотека AWT перестала быть актуальной, т.е. инфраструктура Swing ее полностью вытеснила. Однако подобное предположение неверно. Напротив, понимание AWT все еще важно, потому что AWT лежит в основе Swing, и многие классы AWT прямо или косвенно используются Swing. В итоге для эффективной работы с инфраструктурой Swing по-прежнему требуется твердое знание AWT. Кроме того, в небольших программах, где графический пользовательский интерфейс эксплуатируется лишь минимально, применение AWT может оказаться целесообразным. Таким образом, несмотря на то, что AWT представляет собой старейшую инфраструктуру для построения графических пользовательских интерфейсов Java, базовое владение ее основами важно и на сегодняшний день.

И последнее замечание, прежде чем приступить к обсуждению: библиотека AWT довольно большая, и ее полное описание заняло бы целую книгу, так что подробно рассмотреть буквально каждый класс, метод или переменную экземпляра AWT невозможно. Тем не менее, в этой и последующих главах объясняются основные приемы, необходимые для использования AWT. Затем вы сможете исследовать другие части AWT самостоятельно и вдобавок будете готовы перейти к Swing.

На заметку! Если вы еще не прочитали главу 25, то самое время восполнить пробел. В ней представлен обзор обработки событий, которая задействована во многих примерах текущей главы.

Классы AWT

Классы AWT содержатся в пакете `java.awt` — одном из самых крупных пакетов Java. К счастью, поскольку он логически организован по нисходящей иерархической схеме, его легче понять и использовать, чем может показаться на первый взгляд. Начиная с версии JDK 9, пакет `java.awt` входит в состав модуля `java.desktop`. В табл. 26.1 перечислены избранные классы AWT.

Таблица 26.1. Избранные классы AWT

Класс	Описание
`AWTEvent`	Инкапсулирует события AWT
`AWTEventMulticaster`	Отправляет события множеству прослушивателей
`BorderLayout`	Диспетчер граничной компоновки. Граничная компоновка располагает компоненты в пяти местах: на севере, юге, востоке, западе и по центру
`Button`	Создает элемент управления типа кнопки
`Canvas`	Пустое, свободное от семантики окно
`CardLayout`	Диспетчер карточной компоновки. Карточная компоновка эмулирует индексные карты, из которых видна только верхняя
`Checkbox`	Создает элемент управления типа флажка
`CheckboxGroup`	Создает группу элементов управления типа флажка
`CheckboxMenuItem`	Создает переключаемый пункт меню
`Choice`	Создает раскрывающийся список
`Color`	Управляет цветами в переносимой и не зависящей от платформы манере
`Component`	Абстрактный суперкласс для разнообразных компонентов AWT

Продолжение табл. 26.1

Класс	Описание
`Container`	Подкласс `Component`, который может содержать другие компоненты
`Cursor`	Инкапсулирует растровый курсор
`Dialog`	Создает диалоговое окно
`Dimension`	Указывает размеры объекта. Ширина хранится в `width`, а высота — в `height`
`EventQueue`	Организует очередь событий
`FileDialog`	Создает окно, в котором можно выбрать файл
`FlowLayout`	Диспетчер поточной компоновки, которая позиционирует компоненты слева направо и сверху вниз
`Font`	Инкапсулирует печатный шрифт
`FontMetrics`	Инкапсулирует разнообразную информацию о шрифте, которая помогает отображать текст в окне
`Frame`	Создает стандартное окно, которое имеет строку заголовка, углы, позволяющие изменять размеры, и панель меню
`Graphics`	Инкапсулирует графический контекст, который используется различными методами вывода для отображения выходных данных в окне
`GraphicsDevice`	Описывает графическое устройство, такое как экран или принтер
`GraphicsEnvironment`	Описывает совокупность доступных объектов `Font` и `GraphicsDevice`
`GridBagConstraints`	Определяет разнообразные ограничения, касающиеся класса `GridBagLayout`
`GridBagLayout`	Диспетчер гибкой сеточной компоновки, которая отображает компоненты согласно ограничениям, указанным в `GridBagConstraints`
`GridLayout`	Диспетчер сеточной компоновки, которая отображает компоненты внутри двумерной сетки
`Image`	Инкапсулирует графическое изображение
`Insets`	Инкапсулирует границы контейнера
`Label`	Создает метку, которая отображает строку
`List`	Создает список, из которого пользователь может производить выбор. Похож на стандартное окно со списком в Windows

Окончание табл. 26.1

Класс	Описание
MediaTracker	Управляет медиаобъектом
Menu	Создает раскрывающееся меню
MenuBar	Создает панель меню
MenuComponent	Абстрактный класс, реализуемый различными классами меню
MenuItem	Создает пункт меню
MenuShortcut	Инкапсулирует клавиатурное сокращение для доступа к пункту меню
Panel	Самый простой конкретный подкласс Container
Point	Инкапсулирует пару декартовых координат, хранящихся в x и y
Polygon	Инкапсулирует многоугольник
PopupMenu	Инкапсулирует всплывающее меню
PrintJob	Абстрактный класс, который представляет задание на печать
Rectangle	Инкапсулирует прямоугольник
Robot	Поддерживает автоматизированное тестирование приложений на базе AWT
Scrollbar	Создает элемент управления типа полосы прокрутки
ScrollPane	Контейнер, который предоставляет горизонтальную и/или вертикальную полосу прокрутки для другого компонента
SystemColor	Содержит цвета для виджетов (графических элементов управления) графического пользовательского интерфейса, таких как окна, полосы прокрутки, текст и т.д.
TextArea	Создает элемент управления типа многострочного редактора
TextComponent	Суперкласс для TextArea и TextField
TextField	Создает элемент управления типа однострочного редактора
Toolkit	Абстрактный класс, реализуемый библиотекой AWT
Window	Создает окно, которое не имеет рамки, панели меню и заголовка

Хотя базовая структура библиотеки AWT не менялась со времен Java 1.0, некоторые первоначальные методы устарели и были заменены новыми. Ради обратной совместимости в Java по-прежнему поддерживаются все первоначальные методы из версии Java 1.0, но поскольку они не предназначены для применения в новом коде, в книге они не рассматриваются.

Основы окон

Окна в библиотеке AWT определяются согласно иерархии классов, которая добавляет функциональность и специфичность к каждому уровню. Вероятно, двумя наиболее важными классами, связанными с окнами, можно считать `Frame` и `Panel`. Класс `Frame` инкапсулирует окно верхнего уровня и обычно используется для создания того, что можно было бы считать стандартным окном приложения. Класс `Panel` предоставляет контейнер, куда можно добавлять другие компоненты. (Кроме того, `Panel` является суперклассом для класса `Applet`, который в версии JDK 9 был объявлен нерекомендуемым.) Большая часть функциональных средств `Frame` и `Panel` унаследована от их родительских классов. Таким образом, описание иерархий классов, относящихся к `Frame` и `Panel`, имеет фундаментальное значение для их понимания. На рис. 26.1 показана иерархия классов `Panel` и `Frame`. Давайте рассмотрим каждый из них.

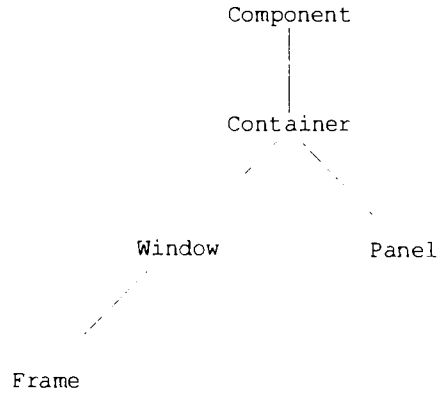


Рис. 26.1. Иерархия классов `Panel` и `Frame`

Component

На вершине иерархии AWT находится `Component` — абстрактный класс, который инкапсулирует все атрибуты визуального компонента. За исключением меню все элементы пользовательского интерфейса, отображаемые на экране и взаимодействующие с пользователем, являются подклассами `Component`. В нем определено более сотни открытых методов, предназначенных для управления событиями вроде ввода с помощью мыши и клавиатуры, позиционирования и изменения размеров окна, а также его перерисовки. Объект

реализации Component отвечает за запоминание текущих цветов переднего плана и фона плюс шрифта, выбранного в данный момент для текста.

Container

Класс Container представляет собой подкласс Component. Он имеет дополнительные методы, которые позволяют вкладывать в него другие объекты Component. Объекты типа Container тоже способны храниться внутри объекта Container (т.к. они сами являются экземплярами Component), что создает многоуровневую систему включения. Контейнер отвечает за компоновку (т.е. позиционирование) любых содержащихся в нем компонентов. Задача решается за счет применения разнообразных диспетчеров компоновки, которые рассматриваются в главе 27.

Panel

Класс Panel — это конкретный подкласс Container. Объект Panel можно считать рекурсивно вкладываемым конкретным экранным компонентом. Добавлять к объекту Panel другие компоненты можно посредством его метода add(), унаследованного от Container. После добавления компонентов их положение и размер можно изменять вручную с использованием методов setLocation(), setSize(), setPreferredSize() или setBounds(), определенных в Component.

Window

Класс Window обеспечивает создание окна верхнего уровня, которое не содержится ни в каком другом объекте, а располагается непосредственно на рабочем столе. Как правило, необходимость в создании объектов Window напрямую не возникает. Взамен будет применяться описанный далее подкласс Window по имени Frame.

Frame

Класс Frame инкапсулирует то, что обычно называют "окном". Он определен как подкласс Window и имеет строку заголовка, панель меню, обрамление и углы, позволяющие изменять размеры. Точный внешний вид объекта Frame отличается в зависимости от среды.

Canvas

Существует еще один класс, который вы сочтете полезным, хотя он не входит в состав иерархии Panel или Frame — Canvas. Унаследованный от Component, класс Canvas инкапсулирует пустое окно, в котором можно рисовать. Пример работы с Canvas будет приведен позже в книге.

Работа с окнами Frame

Обычно чаще всего приходится создавать тип окна приложения на основе AWT, производный от Frame. Как уже упоминалось, такой класс создает окно верхнего уровня в стандартном стиле, которое обладает всеми возможностями, ассоциированными с окном приложения, такими как кнопка закрытия и заголовок. Вот две формы конструктора класса Frame:

```
Frame() throws HeadlessException
Frame(String title) throws HeadlessException
```

Первая форма конструктора создает стандартное окно, не содержащее заголовка, а вторая форма — окно с заголовком, указанным в title. Обратите внимание, что указать размеры окна невозможно. Взамен устанавливать размеры окна придется после его создания. При попытке создать экземпляр Frame в среде, которая не поддерживает взаимодействие с пользователем, генерируется исключение HeadlessException.

Существует несколько основных методов, которые будут использоваться при работе с окнами Frame и рассматриваются в последующих разделах.

Установка размеров окна

Метод setSize() применяется для установки размеров окна:

```
void setSize(int newWidth, int newHeight)
void setSize(Dimension newSize)
```

Новые размеры окна определяется параметрами newWidth и newHeight или полями width и height объекта Dimension, передаваемого в newSize. Размеры задаются в пикселях.

Метод getSize() используется для получения текущих размеров окна. Ниже показана одна из его форм:

```
Dimension getSize()
```

Метод возвращает текущие размеры окна, содержащиеся в полях width и height объекта Dimension.

Скрытие и отображение окна

После создания фреймовое окно не видно до тех пор, пока не будет вызван метод setVisible(). Вот его сигнатура:

```
void setVisible(boolean visibleFlag)
```

Если в visibleFlag передается true, то компонент становится видимым, а если false, тогда он скрывается.

Установка заголовка окна

Изменить заголовок фреймового окна можно с применением метода setTitle(), который имеет следующую общую форму:

```
void setTitle(String newTitle)
```

В newTitle указывается новый заголовок окна.

Закрытие фреймового окна

Когда в программе используется фреймовое окно, после закрытия его потребуется удалить из экрана. Если оно не является окном верхнего уровня приложения, тогда необходимо вызвать метод `setVisible(false)`. В главном окне приложения завершить его работу можно, просто вызвав `System.exit()`, как делалось в примерах из главы 24. Чтобы перехватить событие закрытия окна, понадобится реализовать метод `windowClosing()` интерфейса `WindowListener`. (Интерфейс `WindowListener` обсуждался в главе 25.)

Метод `paint()`

Как было показано в главе 25, вывод в окно обычно происходит, когда исполняющая среда вызывает метод `paint()`, который определен в классе `Component` и переопределяется в классах `Container` и `Window`. Таким образом, он доступен экземплярам `Frame`.

Метод `paint()` вызывается каждый раз, когда вывод приложения на основе AWT должен быть перерисован. Такая ситуация может возникнуть по нескольким причинам. Например, окно программы может быть перекрыто другим окном, после чего снова попасть на передний план. Или же окно может быть свернуто, а затем восстановлено. Метод `paint()` также вызывается при первом отображении окна. Какой бы ни была причина, всякий раз, когда окно должно перерисовать свой вывод, вызывается метод `paint()`. Отсюда вытекает, что программа должна каким-то образом сохранять свой вывод, чтобы его можно было повторно отображать при каждом выполнении `paint()`.

Вот сигнатура метода `paint()`:

```
void paint(Graphics context)
```

Метод `paint()` принимает один параметр типа `Graphics`. Он будет содержать графический контекст, описывающий графическую среду, в которой функционирует программа. Графический контекст используется всякий раз, когда требуется вывод в окно.

Отображение строки

Для вывода строки в экземпляр `Frame` предназначен метод `drawString()`, определенный в классе `Graphics`. Он был представлен в главе 25 и будет широко применяться здесь и в следующей главе. Ниже показана используемая форма:

```
void drawString(String message, int x, int y)
```

В `message` передается строка, которая должна быть выведена, начиная с позиции x,y. Верхний левый угол в окне Java имеет позицию 0,0. Метод `drawString()` не распознает символы новой строки. Если нужно вывести строчку текста в другой строке, то придется делать это вручную, указав точное местоположение X,Y, где строка должна начинаться. (В следующей главе будут продемонстрированы приемы, облегчающие такой процесс.)

Установка цветов фона и переднего плана

Допускается установка цветов фона и переднего плана, которые будут применяться в экземпляре Frame. Метод setBackground() позволяет установить цвет фона, а метод setForeground() — цвет переднего плана (скажем, цвет выводимого текста). Оба метода определены в классе Component и имеют следующие общие формы:

```
void setBackground(Color newColor)
void setForeground(Color newColor)
```

Новый цвет задается в newColor. В классе Color определены перечисленные ниже константы, которые можно использовать для указания цветов (доступны также их версии в верхнем регистре):

```
Color.black                Color.magenta
Color.blue                 Color.orange
Color.cyan                 Color.pink
Color.darkGray             Color.red
Color.gray                 Color.white
Color.green                Color.yellow
Color.lightGray
```

Например, показанный далее код устанавливает цвет фона в зеленый и цвет переднего плана в красный:

```
setBackground(Color.green);
setForeground(Color.red);
```

Кроме того, можно создавать специальные цвета. Удобным местом для первоначальной установки цветов переднего плана и фона является конструктор класса Frame. Разумеется, во время выполнения программы цвета разрешено менять столько раз, сколько необходимо.

Получить текущие значения цветов фона и переднего плана позволяют методы getBackground() и getForeground(). Они тоже определены в Component:

```
Color getBackground()
Color getForeground()
```

Запрос перерисовки

Как правило, приложение осуществляет вывод в свое окно только при вызове его метода paint() исполняющей средой AWT. В результате возникает интересный вопрос: как сама программа может заставить свое окно обновляться с целью отображения нового вывода? Скажем, если программа отображает меняющийся баннер, то какой механизм она применяет для обновления окна каждый раз, когда баннер прокручивается? Вспомните, что одно из фундаментальных архитектурных ограничений, накладываемых на программу с графическим пользовательским интерфейсом, связано с тем, что

она должна быстро возвращать управление исполняющей среде. Например, нельзя организовать цикл внутри paint(), который бы многократно прокручивал баннер, поскольку такой подход предотвратит возврат управления среде AWT. Учитывая упомянутое ограничение, может показаться, что вывод в окно будет в лучшем случае затруднен. К счастью, это не так. Всякий раз, когда вашей программе нужно обновить информацию, отображаемую в ее окне, она просто вызывает метод repaint().

Метод repaint() определен в классе Component. Что касается Frame, то метод repaint() заставляет исполняющую среду AWT выполнять вызов метода update(), который тоже определен в Component. Однако стандартная реализация update() вызывает paint(). Таким образом, для вывода в окно необходимо просто сохранить вывод и затем вызвать repaint(). Далее среда AWT выполнит вызов метода paint(), который сможет отобразить сохраненную информацию. Скажем, если часть программы должна вывести строку, то она может сохранить эту строку в переменной типа String и вызвать метод repaint(). Внутри paint() строка будет выводиться с помощью drawString().

Метод repaint() имеет четыре формы. Давайте рассмотрим их по очереди. Вот простейшая форма repaint():

```
void repaint()
```

Данная форма приводит к перерисовке всего окна. В следующей форме указывается область, которая будет перерисована:

```
void repaint(int left, int top, int width, int height)
```

Координаты левого верхнего угла области задаются в left и top, а ширина и высота области — в width и height. Размеры представлены в пикселях. За счет определения области для перерисовки экономится время, т.к. обновление всего окна требует крупных временных затрат. Если нужно обновить только небольшую часть окна, то более эффективно перерисовывать только эту область.

Вызов repaint() является запросом перерисовки окна в ближайшее время. Тем не менее, если система работает медленно или занята, тогда метод update() может быть вызван не сразу. Несколько запросов перерисовки, которые происходят в течение короткого периода времени, могут быть свернуты средой AWT и метод update() будет вызываться нерегулярно, что приводит к возникновению проблем во многих ситуациях, включая анимацию, которая требует постоянного времени обновления. Одно из решений проблемы предусматривает использование следующих форм repaint():

```
void repaint(long maxDelay)
void repaint(long maxDelay, int x, int y, int width, int height)
```

В maxDelay указывается максимальное количество миллисекунд, которое может пройти до вызова update().

На заметку! Выводить в окно разрешено и в методе, отличающемся от `paint()` или `update()`, для чего он должен получить графический контекст, вызвав `getGraphics()` из `Component`, а затем использовать данный контекст для вывода в окно. Однако в большинстве приложений лучше и проще направлять вывод окна через `paint()` и вызывать `repaint()` при изменении содержимого окна.

Создание приложения на основе `Frame`

Хотя можно просто создать окно, сконструировав экземпляр `Frame`, поступают так редко, потому что созданный подобным образом объект `Frame` не позволяет выполнять многие действия. Например, не будет возможности получать или обрабатывать происходящие в нем события или легко выводить в него информацию.

По указанной причине при построении приложения на основе `Frame` обычно создается подкласс `Frame`. Помимо прочего такой подход позволяет переопределить `paint()` и предоставить обработку событий.

Как было проиллюстрировано в примерах обработки событий из главы 25, построить новое приложение на основе `Frame` фактически довольно легко. В целом понадобится создать подкласс `Frame`, переопределить метод `paint()` для обеспечения вывода в окно и реализовать необходимые прослушиватели событий. Во всех случаях должен быть реализован метод `windowClosing()` интерфейса `WindowListener`. В экземпляре `Frame` верхнего уровня обычно вызывается `System.exit()` для завершения программы. Чтобы просто удалить вторичный экземпляр `Frame` из экрана, можно вызвать `setVisible(false)` при закрытии окна.

После определения подкласса `Frame` можно создать его экземпляр, что даст начало существованию фреймового окна, но оно не будет видимым. Фреймовое окно делается видимым с помощью вызова `setVisible(true)`. При создании окно получает стандартную высоту и ширину. Вызвав метод `setSize()`, можно установить размеры окна явно. Для экземпляра `Frame` верхнего уровня потребуется определить заголовок.

Введение в графику

В состав библиотеки AWT входит несколько методов, поддерживающих графику. Вся графика вычерчивается относительно окна, которым может быть главное окно приложения или дочернее окно. (Такие методы поддерживаются и окнами, основанными на Swing.) Исходная точка каждого окна находится в верхнем левом углу и равна 0,0. Координаты указываются в пикселях. Весь вывод в окно происходит через *графический контекст*.

Графический контекст инкапсулирован в классе `Graphics`. Ниже описаны два способа получения графического контекста:

- он передается методу вроде `paint()` или `update()` в качестве аргумента;
- он возвращается методом `getGraphics()` класса `Component`.

Среди прочих в классе `Graphics` определено несколько методов, которые вычерчивают объекты различных видов, такие как линии, прямоугольники и

дуги. В ряде случаев объекты могут вычерчиваться только по контуру или заполняться. Объекты рисуются и заполняются текущим выбранным цветом — по умолчанию черным. Когда вычерченный графический объект превышает размеры окна, вывод автоматически отсекается. Ниже представлены избранные примеры методов вычерчивания, поддерживаемых классом `Graphics`.

На заметку! Несколько лет назад графические возможности Java были расширены за счет включения новых классов, одним из которых является `Graphics2D`, расширяющий `Graphics`. Класс `Graphics2D` поддерживает ряд улучшений основных возможностей, предлагаемых `Graphics`. Чтобы получить доступ к расширенной функциональности, графический контекст, полученный из метода наподобие `paint()`, потребуется привести к `Graphics2D`. Хотя основных графических функций, поддерживаемых `Graphics`, вполне достаточно для целей настоящей книги, полезно самостоятельно изучить класс `Graphics2D`.

Вычерчивание линий

Линии вычерчиваются с помощью метода `drawLine()`:

```
void drawLine(int startX, int startY, int endX, int endY)
```

Метод `drawLine()` отображает линию с текущим цветом рисования, которая начинается в позиции `startX,startY` и заканчивается в позиции `endX, endY`.

Вычерчивание прямоугольников

Методы `drawRect()` и `fillRect()` отображают контурный и заполненный прямоугольник соответственно:

```
void drawRect(int left, int top, int width, int height)
void fillRect(int left, int top, int width, int height)
```

Верхний левый угол прямоугольника находится в `left,top`. Размеры прямоугольника указываются в `width` и `height`.

Для вычерчивания прямоугольника со скругленными углами предназначены методы `drawRoundRect()` и `fillRoundRect()`:

```
void drawRoundRect(int left, int top, int width, int height,
                   int xDiam, int yDiam)
void fillRoundRect(int left, int top, int width, int height,
                   int xDiam, int yDiam)
```

Верхний левый угол прямоугольника находится в `left,top`. Размеры прямоугольника указываются в `width` и `height`. Диаметр дуги закругления по оси X задается параметром `xDiam`, а по оси Y — параметром `yDiam`.

Вычерчивание эллипсов и окружностей

Метод `drawOval()` предназначен для вычерчивания эллипса, а метод `fillOval()` — для его заполнения:

```
void drawOval(int left, int top, int width, int height)
void fillOval(int left, int top, int width, int height)
```

Эллипс вычерчивается внутри ограничивающего прямоугольника, левый верхний угол которого задается в left и top, а ширина и высота — в width и height. Чтобы нарисовать окружность, необходимо указать квадрат в качестве ограничивающего прямоугольника.

Вычерчивание дуг

Дуги можно вычерчивать с помощью методов drawArc() и fillArc():

```
void drawArc(int left, int top, int width, int height,
             int startAngle, int sweepAngle)
void fillArc(int left, int top, int width, int height,
             int startAngle, int sweepAngle)
```

Дуга ограничена прямоугольником, левый верхний угол которого указан в left и top, а ширина и высота — в width и height. Дуга вычерчивается от startAngle на угловое расстояние, заданное в SweeAngle. Углы представлены в градусах. Ноль градусов соответствует горизонтали в положении часовой стрелки, указывающей на три часа. Дуга рисуется против часовой стрелки при положительном значении sweepAngle и по часовой стрелке при отрицательном значении sweepAngle. Таким образом, чтобы вычертить дугу от двенадцати до шести часов, начальным углом должно быть 90°, а углом развертки — 180°.

Вычерчивание многоугольников

С применением методов drawPolygon() и fillPolygon() можно вычерчивать фигуры произвольной формы:

```
void drawPolygon(int[] x, int[] y, int numPoints)
void fillPolygon(int[] x, int[] y, int numPoints)
```

Конечные точки многоугольника задаются парами координат, содержащимися в массивах x и y. Количество точек, определяемых этими массивами, указывается в параметре numPoints. Существуют альтернативные версии таких методов, в которых многоугольник задается объектом Polygon.

Демонстрация работы методов вычерчивания

Использование описанных выше методов вычерчивания демонстрируется в следующей программе.

```
// Вычерчивание графических элементов.
import java.awt.event.*;
import java.awt.*;
public class GraphicsDemo extends Frame {
  public GraphicsDemo() {
    // Анонимный внутренний класс для обработки событий закрытия окна.
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
```

```
  public void paint(Graphics g) {

    // Вычертить линии.
    g.drawLine(20, 40, 100, 90);
    g.drawLine(20, 90, 100, 40);
    g.drawLine(40, 45, 250, 80);

    // Вычертить прямоугольники.
    g.drawRect(20, 150, 60, 50);
    g.fillRect(110, 150, 60, 50);
    g.drawRoundRect(200, 150, 60, 50, 15, 15);
    g.fillRoundRect(290, 150, 60, 50, 30, 40);

    // Вычертить эллипсы и окружности.
    g.drawOval(20, 250, 50, 50);
    g.fillOval(100, 250, 75, 50);
    g.drawOval(200, 260, 100, 40);

    // Вычертить дуги.
    g.drawArc(20, 350, 70, 70, 0, 180);
    g.fillArc(70, 350, 70, 70, 0, 75);

    // Вычертить многоугольник.
    int[] xpoints = {20, 200, 20, 200, 20};
    int[] ypoints = {450, 450, 650, 650, 450};
    int num = 5;
    g.drawPolygon(xpoints, ypoints, num);
  }

  public static void main(String[] args) {
    GraphicsDemo appwin = new GraphicsDemo();

    appwin.setSize(new Dimension(370, 700));
    appwin.setTitle("GraphicsDemo");
    appwin.setVisible(true);
  }
}
```

На рис. 26.2 показан пример вывода.

Установка размеров графики

Часто требуется привести размеры графического объекта в соответствие с текущими размерами фрейма, в котором рисуется объект. Для этого сначала нужно получить текущие размеры фрейма, вызвав метод `getSize()`. Он возвращает размеры в виде целочисленных значений, хранящихся в полях `width` и `height` экземпляра `Dimension`. Тем не менее, размеры, возвращаемые методом `getSize()`, отражают общий размер фрейма, включая обрамление и строку заголовка. Чтобы получить размеры области для рисования, понадобится уменьшить размер, полученный из `getSize()`, на размеры обрамления/заголовка. Значения, которые описывают размер области обрамления/заголовка, называются *вставками* и получаются вызовом метода `getInsets()`:

```
Insets getInsets()
```

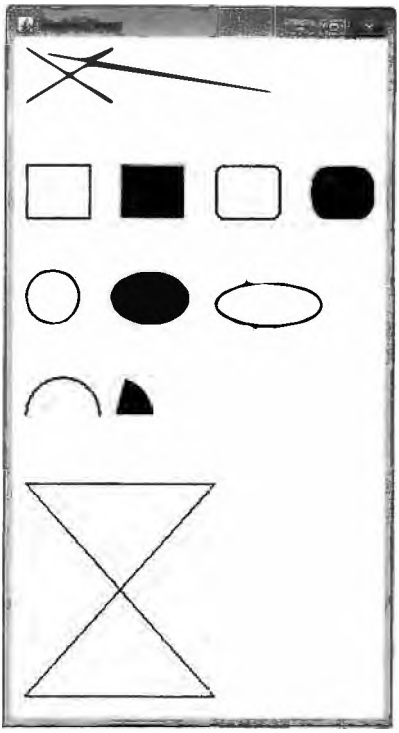


Рис. 26.2. Пример вывода из программы `GraphicsDemo`

Метод `getInsets()` возвращает объект `Insets`, который инкапсулирует размеры вставок в виде четырех значений `int` с именами `left` (левый), `right` (правый), `top` (верхний) и `bottom` (нижний). Следовательно, координатой левого верхнего угла области для рисования будет `left` и `top`, а координатой правого нижнего угла — `width-right` и `height-bottom`. При наличии размеров и вставки можно надлежащим образом масштабировать графический вывод.

Такая методика демонстрируется в приведенной ниже программе, начальный размер фрейма которой имеет 200×200 пикселей. При каждом щелчке кнопкой мыши он увеличивается на 25 пикселей в ширину и в высоту, пока не превысит 500×500. В этот момент следующий щелчок вернет его к размеру 200×200, и процесс начнется заново. Внутри области для рисования вычерчивается крестик, заполняя всю область.

```
// Изменение размеров вывода, чтобы он умещался в текущие размеры окна.
import java.awt.*;
import java.awt.event.*;

public class ResizeMe extends Frame {
  final int inc = 25;
  int max = 500;
  int min = 200;
  Dimension d;
```

```
  public ResizeMe() {
    // Анонимный внутренний класс для обработки событий отпускания кнопки мыши
    addMouseListener(new MouseAdapter() {
      public void mouseReleased(MouseEvent me) {
        int w = (d.width + inc) > max?min :(d.width + inc);
        int h = (d.height + inc) > max?min :(d.height + inc);
        setSize(new Dimension(w, h));
      }
    });
    // Анонимный внутренний класс для обработки событий закрытия окна.
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
  public void paint(Graphics g) {
    Insets i = getInsets();
    d = getSize();
    g.drawLine(i.left, i.top, d.width-i.right, d.height-i.bottom);
    g.drawLine(i.left, d.height-i.bottom, d.width-i.right, i.top);
  }
  public static void main(String[] args) {
    ResizeMe appwin = new ResizeMe();
    appwin.setSize(new Dimension(200, 200));
    appwin.setTitle("ResizeMe");
    appwin.setVisible(true);
  }
}
```

Работа с цветом

Цвета в Java поддерживаются в переносимой и независимой от устройств манере. Цветовая система AWT позволяет указывать любой желаемый цвет. Затем она находит наилучшее соответствие для такого цвета с учетом ограничений оборудования, отвечающего за отображение на устройстве, где в данный момент функционирует программа. Таким образом, в коде не придется учитывать отличия в способах поддержки цвета разными аппаратными устройствами. Цвет инкапсулируется в классе `Color`.

Как упоминалось ранее, в классе `Color` определено несколько констант (скажем, `Color.black`) для указания распространенных цветов. С применением одного из конструкторов `Color` можно создавать также и собственные цвета. Ниже показаны три часто используемых формы конструктора:

```
Color(int red, int green, int blue)
Color(int rgbValue)
Color(float red, float green, float blue)
```

Первая форма конструктора принимает три целых числа, которые указывают цвет в виде смеси красной, зеленой и синей составляющих. Их значения должны находиться в диапазоне от 0 до 255, как в следующем примере:

```
new Color(255, 100, 100);  // светло-красный
```

Вторая форма конструктора принимает одно целочисленное значение, которое содержит смесь красной, зеленой и синей составляющих цвета, упакованную в целое число, причем красная составляющая хранится в битах с 16 по 23, зеленая — в битах с 8 по 15 и синяя — в битах с 0 по 7. Вот пример:

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color darkRed = new Color(newRed);
```

Третья форма конструктора, `Color(float, float, float)`, принимает три значения с плавающей точкой (от `0.0` до `1.0`), определяющие относительные величины красной, зеленой и синей составляющих в смеси.

После создания объект `Color` можно применять для установки цвета переднего плана и/или фона посредством методов `setForeground()` и `setBackground()`. Кроме того, его можно выбрать в качестве текущего цвета рисования.

Методы класса `Color`

В классе `Color` определено несколько методов, помогающих манипулировать цветами. Некоторые из них исследуются ниже.

Использование оттенка, насыщенности и яркости

Цветовая модель HSB (hue-saturation-brightness — оттенок-насыщенность-яркость) является альтернативой модели RGB (red-green-blue — красный-зеленый-синий) для указания конкретных цветов. Образно говоря, *оттенок* представляет собой цветовой круг. Оттенок можно указывать с помощью числа от `0.0` до `1.0`, которое применяется для получения угла в цветовом круге. (Главные цвета приблизительно соответствуют красному, оранжевому, желтому, зеленому, синему, сине-фиолетовому и темно-лиловому.) *Насыщенность* — еще одна шкала в диапазоне от `0.0` до `1.0`, представляющая цвета от светлых пастельных тонов до интенсивных оттенков. Значения *яркости* также варьируются от `0.0` до `1.0`, где `1.0` — яркий белый цвет, а `0.0` — черный. В классе `Color` определены два метода, которые позволяют выполнять преобразования между моделями RGB и HSB:

```
static int HSBtoRGB(float hue, float saturation, float brightness)
static float[] RGBtoHSB(int red, int green, int blue, float[] values)
```

Метод `HSBtoRGB()` возвращает упакованное значение RGB, совместимое с конструктором `Color(int)`. Метод `RGBtoHSB()` возвращает массив типа `float` значений HSB, соответствующих целым числам RGB. Если `values` не равно `null`, тогда этому массиву присваиваются значения HSB и он возвращается. В противном случае создается новый массив и в нем возвращаются значения HSB. В любом случае массив содержит оттенок по индексу 0, насыщенность по индексу 1 и яркость по индексу 2.

getRed(), getGreen(), getBlue()

Красную, зеленую и синюю составляющие цвета можно получать независимо друг от друга с использованием методов getRed(), getGreen() и getBlue():

```
int getRed()
int getGreen()
int getBlue()
```

Каждый метод возвращает в младших восьми битах целочисленного значения компонент цвета RGB, найденный в вызываемом объекте Color.

getRGB()

Для получения упакованного представления цвета RGB предназначен метод getRGB():

```
int getRGB()
```

Возвращаемое значение организовано так, как было описано ранее.

Установка текущего цвета графики

По умолчанию графические объекты рисуются с применением текущего цвета переднего плана, который можно изменить, вызвав метод setColor() класса Graphics:

```
void setColor(Color newColor)
```

В newColor указывается новый цвет для рисования.

Для получения текущего цвета служит метод getColor():

```
Color getColor()
```

Программа, демонстрирующая работу с цветом

В следующей программе создаются цвета, с использованием которых вычерчиваются различные объекты:

```
// Демонстрация работы с цветом.
import java.awt.*;
import java.awt.event.*;

public class ColorDemo extends Frame {

  public ColorDemo() {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  // Вычерчивать с использованием различных цветов.
  public void paint(Graphics g) {
    Color c1 = new Color(255, 100, 100);
```

```
    Color c2 = new Color(100, 255, 100);
    Color c3 = new Color(100, 100, 255);

    g.setColor(c1);
    g.drawLine(20, 40, 100, 100);
    g.drawLine(20, 100, 100, 20);

    g.setColor(c2);
    g.drawLine(40, 45, 250, 180);
    g.drawLine(75, 90, 400, 400);

    g.setColor(c3);
    g.drawLine(20, 150, 400, 40);
    g.drawLine(25, 290, 80, 19);

    g.setColor(Color.red);
    g.drawOval(20, 40, 50, 50);
    g.fillOval(70, 90, 140, 100);

    g.setColor(Color.blue);
    g.drawOval(190, 40, 90, 60);
    g.drawRect(40, 40, 55, 50);

    g.setColor(Color.cyan);
    g.fillRect(100, 40, 60, 70);
    g.drawRoundRect(190, 40, 60, 60, 15, 15);
  }

  public static void main(String[] args) {
    ColorDemo appwin = new ColorDemo();

    appwin.setSize(new Dimension(340, 260));
    appwin.setTitle("ColorDemo");
    appwin.setVisible(true);
  }
}
```

Установка режима рисования

Режим рисования определяет способ вычерчивания объектов в окне. По умолчанию новый вывод в окно перезаписывает любое ранее существовавшее содержимое. Однако метод `setXORMode()` позволяет объединять новые объекты с имеющимся содержимым окна посредством операции исключающего ИЛИ (XOR):

```
void setXORMode(Color xorColor)
```

В `xorColor` указывается цвет, который с помощью операции исключающего ИЛИ будет объединен с содержимым окна при вычерчивании объекта. Преимущество режима с исключающим ИЛИ связано с тем, что он гарантирует видимость нового объекта вне зависимости от того, каким цветом рисуется объект.

Чтобы вернуться к режиму перезаписи, необходимо вызвать метод `setPaintMode()`:

```
void setPaintMode()
```

Обычно режим перезаписи применяется для нормального вывода, а режим с исключающим ИЛИ — для специальных целей. Например, в показанной ниже программе отображается перекрестие, которое отслеживает указатель мыши. Перекрестие объединяется с содержимым окна посредством операции исключающего ИЛИ и будет видимым всегда независимо от цвета содержимого.

```
// Демонстрация режима с исключающим ИЛИ.
import java.awt.*;
import java.awt.event.*;

public class XOR extends Frame {
  int chsX=100, chsY=100;
  public XOR() {
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseMoved(MouseEvent me) {
        int x = me.getX();
        int y = me.getY();
        chsX = x-10;
        chsY = y-10;
        repaint();
      }
    });
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
  // Использовать режим с исключающим ИЛИ.
  public void paint(Graphics g) {
    g.setColor(Color.green);
    g.fillRect(20, 40, 60, 70);
    g.setColor(Color.blue);
    g.fillRect(110, 40, 60, 70);
    g.setColor(Color.black);
    g.fillRect(200, 40, 60, 70);
    g.setColor(Color.red);
    g.fillRect(60, 120, 160, 110);

    // Объединить перекрестие с содержимым посредством
    // операции исключающего ИЛИ.
    g.setXORMode(Color.black);
    g.drawLine(chsX-10, chsY, chsX+10, chsY);
    g.drawLine(chsX, chsY-10, chsX, chsY+10);
    g.setPaintMode();
  }
  public static void main(String[] args) {
    XOR appwin = new XOR();
    appwin.setSize(new Dimension(300, 260));
    appwin.setTitle("XOR Demo");
    appwin.setVisible(true);
  }
}
```

Пример вывода, генерируемого программой, представлен на рис. 26.3.

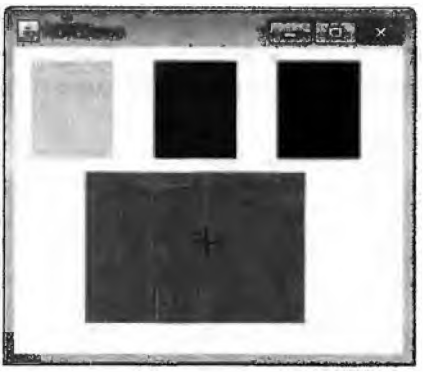


Рис. 26.3. Пример вывода программы, демонстрирующей режим рисования с исключающим ИЛИ

Работа со шрифтами

Библиотека AWT поддерживает множество печатных шрифтов. Шрифты уже давно вышли за рамки традиционного типографского набора и стали важной частью документов и изображений, формируемых на компьютере. Гибкость в AWT достигается за счет абстрагирования операций манипулирования шрифтами и обеспечения динамического выбора шрифтов.

С каждым шрифтом связано название семейства, логическое имя шрифта и наименование гарнитуры. *Название семейства* представляет собой общее имя шрифта, такое как Courier. *Логическое имя* наподобие Monospaced делает возможным связывание с фактическим шрифтом во время выполнения. *Наименование гарнитуры* указывает конкретный шрифт вроде Courier Italic.

Шрифты инкапсулируются в классе `Font`, избранные методы которого кратко описаны в табл. 26.2.

Таблица 26.2. Избранные методы, определенные в классе `Font`

Метод	Описание
`static Font decode (String str)`	Возвращает шрифт с именем, указанным в `str`
`boolean equals (Object FontObj)`	Возвращает `true`, если вызывающий объект содержит такой же шрифт, как и указанный в `FontObj`, или `false` в противном случае
`String getFamily()`	Возвращает название семейства, к которому принадлежит вызывающий шрифт
`static Font getFont (String property)`	Возвращает шрифт, который ассоциирован со свойством системы, указанным в `property`. Если свойство не существует, тогда возвращает `null`

Окончание табл. 26.2

Метод	Описание
`static Font getFont (String property, Font defaultFont)`	Возвращает шрифт, который ассоциирован со свойством системы, указанным в `property`. Если свойство не существует, тогда возвращает шрифт, заданный в `defaultFont`
`String getFontName()`	Возвращает наименование гарнитуры вызывающего шрифта
`String getName()`	Возвращает логическое имя вызывающего шрифта
`int getSize()`	Возвращает размер в пунктах вызывающего шрифта
`int getStyle()`	Возвращает значения начертаний вызывающего шрифта
`int hashCode()`	Возвращает хеш-код, ассоциированный с вызывающим объектом
`boolean isBold()`	Возвращает `true`, если шрифт включает значение начертания `BOLD` (полужирное), или `false` в противном случае
`boolean isItalic()`	Возвращает `true`, если шрифт включает значение начертания `ITALIC` (курсивное), или `false` в противном случае
`boolean isPlain()`	Возвращает `true`, если шрифт включает значение начертания `PLAIN` (обычное), или `false` в противном случае
`String toString()`	Возвращает строковый эквивалент вызывающего шрифта

В классе `Font` определены защищенные переменные, перечисленные в табл. 26.3.

Таблица 26.3. Защищенные переменные, определенные в классе `Font`

Переменная	Описание
`String name`	Имя шрифта
`float pointSize`	Размер шрифта в пунктах
`int size`	Размер шрифта в пунктах
`int style`	Начертание шрифта

Кроме того, имеются также статические поля.

Выяснение доступных шрифтов

При работе со шрифтами часто необходимо знать, какие шрифты доступны на компьютере. Для получения информации такого рода можно использовать метод getAvailableFontFamilyNames(), определенный в классе GraphicsEnvironment:

```
String[] getAvailableFontFamilyNames()
```

Метод возвращает строковый массив, содержащий названия доступных семейств шрифтов. Кроме того, в классе GraphicsEnvironment определен метод getAllFonts():

```
Font[] getAllFonts()
```

Он возвращает массив объектов Font для всех доступных шрифтов.

Поскольку указанные методы являются членами класса GraphicsEnvironment, для их вызова требуется ссылка на GraphicsEnvironment, получить которую можно с применением статического метода getLocalGraphicsEnvironment(), определенного в GraphicsEnvironment:

```
static GraphicsEnvironment getLocalGraphicsEnvironment()
```

Ниже показана программа, демонстрирующая получение названий доступных семейств шрифтов:

```
// Отображение шрифтов.

import java.awt.event.*;
import java.awt.*;

public class ShowFonts extends Frame {
  String msg = "First five fonts: ";
             // Первые пять шрифтов
  GraphicsEnvironment ge;

  public ShowFonts() {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });

    // Получить графическую среду.
    ge = GraphicsEnvironment.getLocalGraphicsEnvironment();

    // Получить список шрифтов.
    String[] fontList = ge.getAvailableFontFamilyNames();

    // Создать строку с именами первых пяти шрифтов.
    for(int i=0; (i < 5) && (i < fontList.length); i++)
      msg += fontList[i] + " ";
  }

  // Отобразить шрифты.
  public void paint(Graphics g) {
    g.drawString(msg, 10, 60);
  }
```

```
  public static void main(String[] args) {
    ShowFonts appwin = new ShowFonts();

    appwin.setSize(new Dimension(500, 100));
    appwin.setTitle("ShowFonts");
    appwin.setVisible(true);
  }
}
```

Пример вывода, генерируемого программой, представлен на рис. 26.4. Имейте в виду, что на своем компьютере вы можете увидеть другой набор шрифтов.

Рис. 26.4. Пример вывода программы, которая получает названия доступных семейств шрифтов

Создание и выбор шрифта

Для создания нового шрифта понадобится сконструировать описывающий его объект `Font`. Конструктор класса `Font` имеет следующую общую форму:

```
Font(String fontName, int fontStyle, int pointSize)
```

В `fontName` передается имя желаемого шрифта. Имя можно указывать с использованием либо названия семейства, либо наименования гарнитуры. Все среды Java будут поддерживать шрифты Dialog, DialogInput, SansSerif, Serif и Monospaced. Шрифт Dialog применяется в диалоговых окнах системы и также используется по умолчанию, если никакой шрифт не был задан явно. Можно задействовать любые другие шрифты, поддерживаемые в конкретной среде, но следует проявлять осторожность, т.к. они могут быть недоступными повсеместно.

Начертание шрифта указывается в параметре `fontStyle`. Оно может включать одну или несколько констант `Font.PLAIN`, `Font.BOLD` и `Font.ITALIC`, для комбинирования которых применяется операция ИЛИ. Скажем, `Font.BOLD | Font.ITALIC` задает полужирное курсивное начертание.

В `pointSize` указывается размер шрифта в пунктах.

Чтобы использовать созданный шрифт, его нужно выбрать с помощью метода `setFont()`, который определен в `Component` и имеет такой общий вид:

```
void setFont(Font fontObj)
```

В `fontObj` передается объект, содержащий желаемый шрифт.

В приведенной далее программе выводятся образцы стандартных шрифтов. Каждый раз, когда совершается щелчок кнопкой мыши в окне программы, выбирается новый шрифт и отображается его имя.

```
// Отображение шрифтов.
import java.awt.*;
import java.awt.event.*;
public class SampleFonts extends Frame {
  int next = 0;
  Font f;
  String msg;
  public SampleFonts() {
    f = new Font("Dialog", Font.PLAIN, 12);
    msg = "Dialog";
    setFont(f);
    addMouseListener(new MyMouseAdapter(this));
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
  public void paint(Graphics g) {
    g.drawString(msg, 10, 60);
  }
  public static void main(String[] args) {
    SampleFonts appwin = new SampleFonts();
    appwin.setSize(new Dimension(200, 100));
    appwin.setTitle("SampleFonts");
    appwin.setVisible(true);
  }
}
class MyMouseAdapter extends MouseAdapter {
  SampleFonts sampleFonts;
  public MyMouseAdapter(SampleFonts sampleFonts) {
    this.sampleFonts = sampleFonts;
  }
  public void mousePressed(MouseEvent me) {
    // Переключать шрифты при каждом щелчке кнопкой мыши.
    sampleFonts.next++;
    switch(sampleFonts.next) {
      case 0:
        sampleFonts.f = new Font("Dialog", Font.PLAIN, 12);
        sampleFonts.msg = "Dialog";
        break;
      case 1:
        sampleFonts.f = new Font("DialogInput", Font.PLAIN, 12);
        sampleFonts.msg = "DialogInput";
        break;
      case 2:
        sampleFonts.f = new Font("SansSerif", Font.PLAIN, 12);
        sampleFonts.msg = "SansSerif";
        break;
```

```
      case 3:
        sampleFonts.f = new Font("Serif", Font.PLAIN, 12);
        sampleFonts.msg = "Serif";
        break;
      case 4:
        sampleFonts.f = new Font("Monospaced", Font.PLAIN, 12);
        sampleFonts.msg = "Monospaced";
        break;
    }
    if(sampleFonts.next == 4) sampleFonts.next = -1;
    sampleFonts.setFont(sampleFonts.f);
    sampleFonts.repaint();
  }
}
```

Пример вывода из этой программы показан на рис. 26.5.

Рис. 26.5. Пример вывода программы, отображающей шрифты

Получение информации о шрифте

Предположим, что необходимо получить информацию о шрифте, выбранном в текущий момент. Для этого сначала должен быть получен текущий шрифт вызовом метода `getFont()`, который определен в классе `Graphics`:

```
Font getFont()
```

После получения шрифта, выбранного в текущий момент, с применением разнообразных методов, определенных в `Font`, можно получить информацию о нем. Например, в следующей программе отображается имя, семейство, размер и начертание выбранного в данный момент шрифта:

```
// Отображение информации о шрифте.

import java.awt.event.*;
import java.awt.*;

public class FontInfo extends Frame {
  public FontInfo() {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
```

```
  public void paint(Graphics g) {
    Font f = g.getFont();
    String fontName = f.getName();
    String fontFamily = f.getFamily();
    int fontSize = f.getSize();
    int fontStyle = f.getStyle();
    String msg = "Family: " + fontName;
              // Семейство
    msg += ", Font: " + fontFamily;
           // Шрифт
    msg += ", Size: " + fontSize + ", Style: ";
           // Размер                Начертание
    if((fontStyle & Font.BOLD) == Font.BOLD)
      msg += "Bold ";
           // Полужирное
    if((fontStyle & Font.ITALIC) == Font.ITALIC)
      msg += "Italic ";
          // Курсивное
    if((fontStyle & Font.PLAIN) == Font.PLAIN)
      msg += "Plain ";
           // Обычное
    g.drawString(msg, 10, 60);
  }
  public static void main(String[] args) {
    FontInfo appwin = new FontInfo();
    appwin.setSize(new Dimension(300, 100));
    appwin.setTitle("FontInfo");
    appwin.setVisible(true);
  }
}
```

Управление выводом текста с использованием FontMetrics

Как объяснялось выше, в Java поддерживается ряд шрифтов. В большинстве шрифтов символы не имеют одинаковые размеры, а потому такие шрифты пропорциональны. Кроме того, высота каждого символа, длина *нижних выносных элементов* (висящих частей букв, таких как у) и расстояние между горизонтальными линиями варьируются от шрифта к шрифту. Вдобавок размер шрифта в пунктах может меняться. Регулируемость упомянутых (и других) атрибутов вряд ли играла бы большую роль, если б не тот факт, что Java требует от программистов ручного управления практически всем выводом текста.

С учетом того, что размер каждого шрифта может отличаться, а шрифты могут изменяться во время выполнения программы, должен существовать какой-нибудь способ выяснения размеров и других атрибутов шрифта, выбранного в текущий момент. Скажем, вывод одной строки текста после другой означает необходимость в наличии способа узнать высоту шрифта и ко-

личество пикселей между строками. Для удовлетворения такой потребности в библиотеке AWT предусмотрен класс `FontMetrics`, который инкапсулирует разнообразную информацию о шрифте. В табл. 26.4. приведена общая терминология, используемая при описании шрифтов.

Таблица 26.4. Общая терминология, применяемая для описания шрифтов

Термин	Описание
Высота	Размер строки текста сверху донизу
Базовая линия	Линия, по которой выравниваются нижние части символов (не считая спуска)
Подъем	Расстояние между базовой линией и верхом символа
Спуск	Расстояние между базовой линией и низом символа
Интерлиньяж	Расстояние между низом одной строки текста и верхом следующей

Вы наверняка заметили, что во многих предшествующих примерах использовался метод `drawString()`, который рисует строку, начиная с указанной позиции, с применением текущего шрифта и цвета. Тем не менее, позиция находится на левом краю базовой линии символов, а не в верхнем левом углу, как обычно бывает при использовании других методов рисования. Распространенной ошибкой является рисование строки с указанием таких же координат, как при рисовании прямоугольника. Например, в случае вычерчивания прямоугольника в верхнем левом углу отобразится полный прямоугольник. Но если в этом месте нарисовать строку `"Typesetting"`, то будут видны только нижние части символов `y`, `p` и `g`. Вы увидите, что с применением метрик шрифта можно правильно определить размещение каждой отображаемой строки.

В классе `FontMetrics` определены методы, помогающие управлять выводом текста. Некоторые часто используемые методы кратко описаны в табл. 26.5. Они содействуют корректному отображению текста в окне.

Таблица 26.5. Избранные методы, определенные в классе `FontMetrics`

Метод	Описание
`int bytesWidth(byte[] b, int start, int numBytes)`	Возвращает ширину `numBytes` символов из массива `b`, начиная с элемента `start`
`int charsWidth(char[] c, int start, int numChars)`	Возвращает ширину `numChars` символов из массива `c`, начиная с элемента `start`
`int charWidth(char c)`	Возвращает ширину символа `c`
`int charWidth(int c)`	Возвращает ширину символа `c`

Окончание табл. 26.5

Метод	Описание
`int getAscent()`	Возвращает подъем шрифта
`int getDescent()`	Возвращает спуск шрифта
`Font getFont()`	Возвращает шрифт
`int getHeight()`	Возвращает высоту строки текста. Это значение можно использовать для многострочного вывода текста в окне
`int getLeading()`	Возвращает расстояние между строками текста (т.е. интерлиньяж)
`int getMaxAdvance()`	Возвращает оценочную ширину самого широкого символа или –1, если это значение недоступно
`int getMaxAscent()`	Возвращает максимальный подъем шрифта
`int getMaxDescent()`	Возвращает максимальный спуск шрифта
`int[] getWidths()`	Возвращает ширину первых 256 символов
`int stringWidth(String str)`	Возвращает ширину строки, указанной в `str`
`String toString()`	Возвращает строковый эквивалент вызывающего объекта

Вероятно, класс `FontMetrics` чаще всего применяется для определения расстояния между строками текста. Второй распространенный сценарий использования касается выяснения длины отображаемой строки. Ниже вы узнаете, как решать такие задачи.

В общем случае для отображения нескольких строк текста в программе придется вручную отслеживать текущую позицию вывода. Каждый раз, когда требуется новая строка, координата Y должна быть перемещена в начало следующей строки. Каждый раз, когда отображается строка, координата X должна быть установлена в точку, где строка заканчивается. В итоге следующую строку можно вывести так, чтобы она начиналась в конце предыдущей строки.

Определить расстояние между строками позволит значение, возвращаемое методом `getLeading()`. Для определения общей высоты шрифта понадобится добавить значение, возвращаемое `getAscent()`, к значению, возвращаемому `getDescent()`. Затем полученные значения можно применять для позиционирования каждой строки выводимого текста. Однако во многих случаях использовать эти отдельные значения не придется. Часто нужно знать лишь общую высоту строки, которая представляет собой сумму значений интерлиньяжа, подъема и спуска шрифта. Получить такое значение легче всего, вы-

звав метод getHeight(). Во многих случаях можно просто увеличивать координату Y на это значение каждый раз, когда при выводе текста необходимо переходить на следующую строку.

Чтобы начать вывод с места, где был закончен предыдущий вывод, в той же самой строке, нужно знать длину в пикселях каждой отображаемой строки, для получения которой потребуется вызвать метод stringWidth(). Возвращенное значение можно применять для продвижения координаты X при каждом отображении строки.

В следующей программе демонстрируется многострочный вывод текста в окно. В ней также отображается несколько предложений в одной строке. Обратите внимание на переменные curX и curY, которые отслеживают текущую позицию вывода текста.

```
// Демонстрация многострочного вывода текста.

import java.awt.event.*;
import java.awt.*;

public class MultiLine extends Frame {
  int curX=20, curY=40;           // текущая позиция

  public MultiLine() {
    Font f = new Font("SansSerif", Font.PLAIN, 12);
    setFont(f);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    nextLine("This is on line one.", g);
    nextLine("This is on line two.", g);
    sameLine(" This is on same line.", g);
    sameLine(" This, too.", g);
    nextLine("This is on line three.", g);

    curX = 20; curY = 40; //сбрасывать координаты при каждом перерисовывании
  }

  // Advance to next line.
  void nextLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    curY += fm.getHeight();       // перейти на следующую строку
    curX = 20;
    g.drawString(s, curX, curY);
    curX += fm.stringWidth(s);    // перейти в конец строки
  }

  // Отобразить в той же строке.
  void sameLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();
```

```
    g.drawString(s, curX, curY);
    curX += fm.stringWidth(s);          // перейти в конец строки
  }

  public static void main(String[] args) {
    MultiLine appwin = new MultiLine();
    appwin.setSize(new Dimension(300, 120));
    appwin.setTitle("MultiLine");
    appwin.setVisible(true);
  }
}
```

Пример вывода, генерируемого программой, представлен на рис. 26.6.

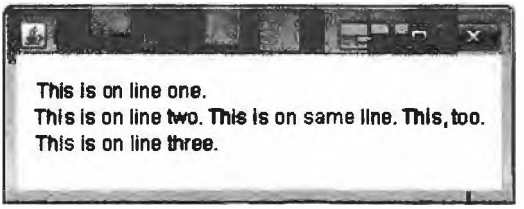


Рис. 26.6. Пример вывода программы, отображающей многострочный текст

ГЛАВА

27 Использование элементов управления, диспетчеров компоновки и меню AWT

В этой главе продолжается обзор библиотеки AWT. Сначала исследуются некоторые элементы управления и диспетчеры компоновки AWT, а затем рассматриваются меню и панель меню. В главу также включено обсуждение диалогового окна.

Элементы управления представляют собой компоненты, которые позволяют пользователю взаимодействовать с приложением разнообразными способами — например, часто используемым элементом управления является кнопка. *Диспетчер компоновки* автоматически позиционирует компоненты внутри контейнера. Таким образом, внешний вид окна определяется комбинацией содержащихся в нем элементов управления и диспетчера компоновки, применяемого для их позиционирования.

Помимо элементов управления фреймовое окно может содержать *панель меню* в стандартном стиле. Каждый пункт в панели меню активизирует раскрывающееся меню вариантов, доступных для выбора пользователем. Оно образует *главное меню* приложения. Как правило, панель меню располагается в верхней части окна. Несмотря на различия во внешнем виде, панели меню обрабатываются во многом так же, как и другие элементы управления.

Хотя позиционировать компоненты в окне можно и вручную, делать это довольно утомительно. Диспетчер компоновки автоматизирует такую задачу. В первой части главы, посвященной различным элементам управления, будет использоваться простой диспетчер компоновки, который отображает компоненты в контейнере с построчной организацией сверху вниз. После исследования элементов управления будут обсуждаться другие диспетчеры компоновки, что позволит вам понять, каким образом лучше управлять расположением элементов управления.

Прежде чем продолжить, важно подчеркнуть, что в настоящее время вам редко придется создавать графические пользовательские интерфейсы, основанные целиком на AWT, поскольку для Java были разработаны более мощные инфраструктуры подобного рода. Однако предложенный здесь материал остается важным по нескольким причинам. Во-первых, большинство сведений и многие методики, связанные с элементами управления и обработкой событий, можно обобщить на Swing. (Как упоминалось в предыдущей главе,

инфраструктура Swing построена на основе AWT.) Во-вторых, описанные здесь диспетчеры компоновки применяются также и в Swing. В-третьих, для ряда небольших приложений компоненты AWT могут оказаться самым подходящим вариантом. Наконец, вы можете столкнуться с устаревшим кодом, где используется AWT. Следовательно, базовое понимание библиотеки AWT важно для всех программистов на Java.

Основы элементов управления AWT

В библиотеке AWT поддерживаются элементы управления следующих видов:

- метки;
- кнопки;
- флажки;
- списки выбора;
- списки;
- полосы прокрутки;
- элементы для редактирования текста.

Все элементы управления AWT определены как подклассы `Component`. Несмотря на то что набор элементов управления, предлагаемый библиотекой AWT, не особенно богат, его вполне достаточно для простых приложений вроде небольших служебных программ, предназначенных для собственных нужд. Он также весьма полезен для ознакомления с базовыми концепциями и методиками обработки событий в элементах управления. Тем не менее, важно отметить, что Swing предоставляет значительно более крупный и сложный набор элементов управления, который лучше подходит для создания коммерческих приложений.

Добавление и удаление элементов управления

Чтобы включить элемент управления в состав окна, его потребуется поместить в окно, для чего сначала создать экземпляр желаемого элемента управления и затем добавить в окно, вызвав метод `add()`, который определен в классе `Container`. Метод `add()` имеет несколько форм. Вот форма, применяемая в первой части главы:

```
Component add(Component compRef)
```

В `compRef` передается ссылка на экземпляр элемента управления, который необходимо добавить. Возвращается ссылка на данный объект. После добавления элемента управления он автоматически становится видимым всякий раз, когда отображается его родительское окно.

Иногда может понадобиться удалить элемент управления из окна, когда он больше не нужен. Для этого следует вызвать метод `remove()`, который тоже определен в классе `Container`. Ниже показана одна из его форм:

```
void remove(Component compRef)
```

В `compRef` передается ссылка на экземпляр элемента управления, который должен быть удален. Для удаления всех элементов управления предназначен метод `removeAll()`.

Реагирование на события, генерируемые элементами управления

Кроме меток, которые являются пассивными, все остальные элементы управления генерируют события при доступе к ним со стороны пользователя. Например, когда пользователь щелкает на кнопке, отправляется событие, идентифицирующее кнопку. В общем случае программа просто реализует подходящий интерфейс, а затем регистрирует прослушиватель событий для каждого элемента управления, который необходимо отслеживать. Как объяснялось в главе 25, после установки прослушивателя ему автоматически отправляются события. В последующих разделах указан соответствующий интерфейс для каждого элемента управления.

Исключение `HeadlessException`

Большинство описанных в главе элементов управления AWT имеют конструкторы, которые могут генерировать исключение `HeadlessException` при попытке создать экземпляр компонента графического пользовательского интерфейса в среде, не обладающей интерактивностью (скажем, в среде, где нет дисплея, мыши или клавиатуры). Упомянутое исключение можно использовать для написания кода, который способен адаптироваться к неинтерактивным средам. (Конечно, это возможно далеко не всегда.) Исключение `HeadlessException` не обрабатывается в рассматриваемых в главе примерах программ, т.к. для демонстрации элементов управления AWT требуется интерактивная среда.

Метки

Самым простым элементом управления является *метка*, представляющая собой объект типа `Label`, который содержит отображаемую строку. Метки — пассивные элементы управления, которые не поддерживают никакого взаимодействия с пользователем. В классе `Label` определены следующие конструкторы:

```
Label() throws HeadlessException
Label(String str) throws HeadlessException
Label(String str, int how) throws HeadlessException
```

Первая форма конструктора создает пустую метку. Вторая форма создает метку, которая содержит строку, заданную параметром `str`. Строка выравнивается по левому краю. Третья форма создает метку, которая содержит строку, указанную в `str`, с применением выравнивания, заданного в `how`. Значением параметра `how` должна быть одна из трех констант: `Label.LEFT`, `Label.RIGHT` или `Label.CENTER`.

С помощью метода `setText()` можно установить или изменить текст в метке, а посредством метода `getText()` получить текущую метку:

```
void setText(String str)
String getText()
```

При вызове `setText()` в `str` указывается новая метка. Вызов `getText()` возвращает текущую метку. С использованием метода `setAlignment()` можно установить выравнивание строки внутри метки.

Для получения текущего выравнивания понадобится вызвать метод `getAlignment()`:

```
void setAlignment(int how)
int getAlignment()
```

Значением `how` должна быть одна из указанных ранее констант выравнивания. В приведенном далее примере создаются три метки, которые добавляются к объекту `Frame`:

```
// Демонстрация использования меток.
import java.awt.*;
import java.awt.event.*;

public class LabelDemo extends Frame {
  public LabelDemo() {

    // Использовать поточную компоновку.
    setLayout(new FlowLayout());
    Label one = new Label("One");
    Label two = new Label("Two");
    Label three = new Label("Three");

    // Добавить метки во фрейм.
    add(one);
    add(two);
    add(three);
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
  public static void main(String[] args) {
    LabelDemo appwin = new LabelDemo();
    appwin.setSize(new Dimension(300, 100));
    appwin.setTitle("LabelDemo");
    appwin.setVisible(true);
  }
}
```

На рис. 27.1 показан пример вывода из программы LabelDemo.

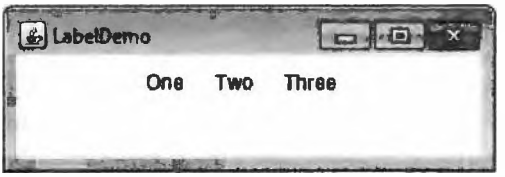


Рис. 27.1. Пример вывода из программы LabelDemo

Обратите внимание, что метки располагаются в окне слева направо, что автоматически обрабатывается диспетчером компоновки FlowLayout, который входит в набор таких диспетчеров, предоставляемых AWT. Здесь он применяется со своей стандартной конфигурацией, которая предусматривает размещение компонентов построчно, слева направо, сверху вниз и по центру. Как будет показано далее в главе, диспетчер компоновки FlowLayout поддерживает несколько параметров, но пока вполне достаточно его стандартного поведения. Обратите внимание, что FlowLayout выбирается в качестве диспетчера компоновки с помощью метода setLayout(), устанавливающего диспетчер компоновки, ассоциированный с контейнером, которым в данном случае является объемлющий фрейм. Хотя FlowLayout очень удобен и приемлем для текущих целей, он не позволяет управлять размещением компонентов в окне. После детального рассмотрения темы диспетчеров компоновки позже в главе вы узнаете, как получить больший контроль над организацией своих окон.

Использование кнопок

Пожалуй, наиболее широко применяемым элементом управления можно считать *кнопку* — компонент, который содержит метку и генерирует событие при щелчке на нем. Кнопки представляют собой объекты типа Button. В классе Button определены два конструктора:

```
Button() throws HeadlessException
Button(String str) throws HeadlessException
```

Первая форма конструктора создает кнопку без метки, а вторая форма — кнопку с меткой, указанной в str.

После создания кнопки установить ее метку можно с помощью метода setLabel(), а получить — посредством метода getLabel():

```
void setLabel(String str)
String getLabel()
```

В str указывается новая метка кнопки.

Обработка событий для кнопок

При каждом щелчке на кнопке генерируется событие действия, которое отправляется всем прослушивателям, ранее зарегистрированным для

получения уведомлений о событиях действий от данного компонента. Прослушиватели реализуют интерфейс ActionListener, в котором определен метод actionPerformed(), вызываемый при возникновении события. В качестве аргумента методу передается объект ActionEvent, который содержит как ссылку на кнопку, сгенерировавшую событие, так и ссылку на *строку с командой действия*, ассоциированную с кнопкой. По умолчанию строка с командой действия является меткой кнопки. Для идентификации кнопки можно использовать либо ссылку на кнопку, либо строку с командой действия. (Вскоре вы увидите примеры применения каждого подхода.)

В приведенном далее примере создаются три кнопки с метками "Yes" (Да), "No" (Нет) и "Undecided" (В раздумьях). Щелчок на одной из них инициирует отображение сообщения о том, на какой кнопке он был совершен. В этой версии команда действия кнопки (которой по умолчанию является ее метка) используется для определения того, на какой кнопке делался щелчок. Для получения метки необходимо вызвать метод getActionCommand() на объекте ActionEvent, переданном в actionPerformed().

```
// Демонстрация использования кнопок.

import java.awt.*;
import java.awt.event.*;

public class ButtonDemo extends Frame implements ActionListener {
  String msg = "";
  Button yes, no, maybe;

  public ButtonDemo() {

    // Использовать поточную компоновку.
    setLayout(new FlowLayout());

    // Создать несколько кнопок.
    yes = new Button("Yes");           // Да
    no = new Button("No");             // Нет
    maybe = new Button("Undecided");   // В раздумьях

    // Добавить кнопки во фрейм.
    add(yes);
    add(no);
    add(maybe);

    // Добавить прослушиватели событий действий для кнопок.
    yes.addActionListener(this);
    no.addActionListener(this);
    maybe.addActionListener(this);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  // Обработать события действий для кнопок.
  public void actionPerformed(ActionEvent ae) {
    String str = ae.getActionCommand();
```

```
    if(str.equals("Yes")) {
      msg = "You pressed Yes.";
          // Щелчок на кнопке Yes
    }
    else if(str.equals("No")) {
      msg = "You pressed No.";
          // Щелчок на кнопке No
    }
    else {
      msg = "You pressed Undecided.";
          // Щелчок на кнопке Undecided
    }
    repaint();
  }
  public void paint(Graphics g) {
    g.drawString(msg, 20, 100);
  }
  public static void main(String[] args) {
    ButtonDemo appwin = new ButtonDemo();
    appwin.setSize(new Dimension(250, 150));
    appwin.setTitle("ButtonDemo");
    appwin.setVisible(true);
  }
}
```

На рис. 27.2 показан пример вывода из программы ButtonDemo.

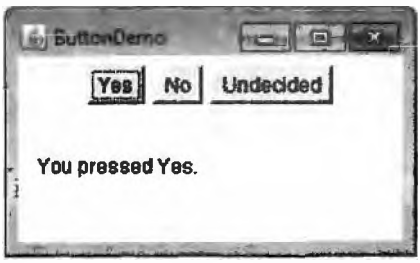


Рис. 27.2. Пример вывода из программы ButtonDemo

Ранее уже упоминалось, что кроме сравнения строк с командами действий выяснить, на какой кнопке был произведен щелчок, можно за счет сравнения объекта, полученного из метода getSource(), с объектами кнопок, которые были добавлены в окно. Для этого необходимо вести список добавляемых объектов. Такой подход демонстрируется в следующей программе:

```
// Распознавание объектов Button.

import java.awt.*;
import java.awt.event.*;

public class ButtonList extends Frame implements ActionListener {
  String msg = "";
  Button[] bList = new Button[3];
```

```
  public ButtonList() {
    // Использовать поточную компоновку.
    setLayout(new FlowLayout());
    // Создать несколько кнопок.
    Button yes = new Button("Yes");
    Button no = new Button("No");
    Button maybe = new Button("Undecided");
    // Сохранить ссылки на добавленные кнопки.
    bList[0] = (Button) add(yes);
    bList[1] = (Button) add(no);
    bList[2] = (Button) add(maybe);
    // Зарегистрировать прослушиватели для получения событий действий.
    for(int i = 0; i < 3; i++) {
      bList[i].addActionListener(this);
    }
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
  // Обработать события действий для кнопок.
  public void actionPerformed(ActionEvent ae) {
    for(int i = 0; i < 3; i++) {
      if(ae.getSource() == bList[i]) {
        msg = "You pressed " + bList[i].getLabel();
          // Щелчок на кнопке ...
      }
    }
    repaint();
  }
  public void paint(Graphics g) {
    g.drawString(msg, 20, 100);
  }
  public static void main(String[] args) {
    ButtonList appwin = new ButtonList();
    appwin.setSize(new Dimension(250, 150));
    appwin.setTitle("ButtonList");
    appwin.setVisible(true);
  }
}
```

В данной версии программы при добавлении кнопки во фрейм ссылка на нее сохраняется в массиве. (Вспомните, что метод `add()` возвращает ссылку на добавляемую кнопку.) Внутри метода `actionPerformed()` с применением этого массива выясняется, на какой кнопке был совершен щелчок.

В простых программах распознавать кнопки обычно легче по их меткам. Однако если планируется изменение меток внутри кнопок во время выпол-

нения или в программе используются кнопки с одинаковыми метками, то может оказаться проще определить, на какой кнопке был произведен щелчок, с применением ссылки на ее объект. Кроме того, строку с командой действия кнопки можно установить в значение, отличающееся от ее метки, с помощью метода `setActionCommand()`, который изменяет строку с командой действия, но не влияет на строку, используемую для метки кнопки. Таким образом, установка строки с командой действия позволяет различать команду действия и метку кнопки.

В ряде случаев обрабатывать события действий, генерируемые кнопкой (или другим типом элемента управления), можно с применением анонимного внутреннего класса (как было описано в главе 25) или лямбда-выражения (обсуждаемого в главе 15). Скажем, вот набор обработчиков событий действий для предшествующих программ, которые используют лямбда-выражения:

```
// Использовать лямбда-выражения для обработки событий действий.
yes.addActionListener((ae) -> {
  msg = "You pressed " + ae.getActionCommand();
  repaint();
});

no.addActionListener((ae) -> {
  msg = "You pressed " + ae.getActionCommand();
  repaint();
});

maybe.addActionListener((ae) -> {
  msg = "You pressed " + ae.getActionCommand();
  repaint();
});
```

Код работает по той причине, что в функциональном интерфейсе `ActionListener` определен ровно один абстрактный метод и потому его можно применять в лямбда-выражении. Как правило, лямбда-выражение можно использовать для обработки события AWT, когда его прослушиватель реализует функциональный интерфейс. Например, `ItemListener` тоже является функциональным интерфейсом. Разумеется, что именно будет применяться — традиционный подход, анонимный внутренний класс или лямбда-выражение, — зависит от природы разрабатываемого приложения.

Использование флажков

Флажок — это элемент управления, который используется для включения или отключения возможного варианта. Он состоит из небольшого окошка, которое может содержать или не содержать галочку. С каждым флажком ассоциирована метка, описывающая вариант, который представлен флажком. Чтобы изменить состояние флажка, необходимо щелкнуть на нем. Флажки можно применять индивидуально или в составе группы. Флажки являются объектами класса `Checkbox`.

Класс `Checkbox` поддерживает перечисленные далее конструкторы:

```
Checkbox() throws HeadlessException
Checkbox(String str) throws HeadlessException
Checkbox(String str, boolean on) throws HeadlessException
Checkbox(String str, boolean on, CheckboxGroup cbGroup)
  throws HeadlessException
Checkbox(String str, CheckboxGroup cbGroup, boolean on)
  throws HeadlessException
```

Первая форма конструктора создает флажок с изначально пустой меткой и неотмеченным состоянием. Вторая форма создает флажок с меткой, указанной в `str`, и неотмеченным состоянием. Третья форма конструктора позволяет установить начальное состояние флажка. Если `on` имеет значение `true`, то флажок изначально отмечен; иначе он очищен. Четвертая и пятая формы создают флажок, метка которого указывается в `str`, а группа — в `cbGroup`. Если флажок не входит в состав группы, тогда в `cbGroup` должно передаваться значение `null`. (Группы флажков описаны в следующем разделе.) Значение `on` определяет начальное состояние флажка.

Для получения текущего состояния флажка понадобится вызвать метод `getState()`. Для установки состояния флажка необходимо вызвать метод `setState()`. Получить текущую метку, ассоциированную с флажком, можно посредством метода `getLabel()`. Чтобы установить метку, нужно вызвать метод `setLabel()`. Вот как выглядят упомянутые методы:

```
boolean getState()
void setState(boolean on)
String getLabel()
void setLabel(String str)
```

Если `on` имеет значение `true`, то флажок отмечен, а если `false`, тогда флажок очищен. Строка, указанная в `str`, становится новой меткой для вызывающего флажка.

Обработка событий для флажков

При каждой отметке или снятии отметки с флажка генерируется событие элемента. Оно отправляется всем прослушивателям, которые ранее зарегистрировались для получения уведомлений о событиях элемента от данного компонента. Все прослушиватели реализуют интерфейс `ItemListener`, в котором определен метод `itemStateChanged()`. В качестве аргумента методу `itemStateChanged()` предоставляется объект `ItemEvent`, содержащий информацию о событии (например, производился выбор или отмена выбора).

В следующей программе создаются четыре флажка, первый из которых изначально отмечен. Состояние всех флажков отображается. При изменении состояния любого флажка отображение состояния обновляется.

```
// Демонстрация использования флажков.

import java.awt.*;
import java.awt.event.*;

public class CheckboxDemo extends Frame implements ItemListener {
  String msg = "";
```

```
  Checkbox windows, android, linux, mac;

  public CheckboxDemo() {

    // Использовать поточную компоновку.
    setLayout(new FlowLayout());

    // Создать флажки.
    windows = new Checkbox("Windows", true);
    android = new Checkbox("Android");
    linux = new Checkbox("Linux");
    mac = new Checkbox("Mac OS");

    // Добавить флажки во фрейм.
    add(windows);
    add(android);
    add(linux);
    add(mac);

    // Добавить прослушиватели событий элементов.
    windows.addItemListener(this);
    android.addItemListener(this);
    linux.addItemListener(this);
    mac.addItemListener(this);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Отобразить текущее состояние флажков.
  public void paint(Graphics g) {
    msg = "Current state: ";
      // Текущее состояние
    g.drawString(msg, 20, 120);
    msg = "   Windows: " + windows.getState();
    g.drawString(msg, 20, 140);
    msg = "   Android: " + android.getState();
    g.drawString(msg, 20, 160);
    msg = "   Linux: " + linux.getState();
    g.drawString(msg, 20, 180);
    msg = "   Mac OS: " + mac.getState();
    g.drawString(msg, 20, 200);
  }

  public static void main(String[] args) {
    CheckboxDemo appwin = new CheckboxDemo();

    appwin.setSize(new Dimension(240, 220));
    appwin.setTitle("CheckboxDemo");
    appwin.setVisible(true);
  }
}
```

Пример вывода из программы CheckboxDemo показан на рис. 27.3.

Рис. 27.3. Пример вывода из программы CheckboxDemo

Группы флажков

Существует возможность создать набор взаимоисключающих флажков, где в любой момент времени может быть отмечен один и только один флажок в группе. Такие флажки часто называют *переключателями*, потому что они действуют подобно селектору станций в радиоприемнике, который в каждый конкретный момент допускает выбор только одной станции. Для создания набора взаимоисключающих флажков сначала необходимо определить группу, к которой они будут принадлежать, и затем указать эту группу при создании флажков. Группы флажков представляют собой объекты типа CheckboxGroup. В классе CheckboxGroup определен лишь стандартный конструктор, создающий пустую группу.

Определить, какой флажок в группе отмечен в данный момент, можно с помощью вызова метода getSelectedCheckbox(). Чтобы отметить флажок, понадобится вызвать метод setSelectedCheckbox(). Вот как выглядят упомянутые методы:

```
Checkbox getSelectedCheckbox()
void setSelectedCheckbox(Checkbox which)
```

В which указывается флажок, который нужно отметить. Ранее отмеченный флажок утрачивает отметку.

Ниже приведена программа, в которой используются флажки, входящие в группу:

```
// Демонстрация группы флажков.

import java.awt.*;
import java.awt.event.*;

public class CBGroup extends Frame implements ItemListener {
  String msg = "";
  Checkbox windows, android, linux, mac;
  CheckboxGroup cbg;
```

```
  public CBGroup() {
    // Использовать поточную компоновку.
    setLayout(new FlowLayout());
    // Создать группу флажков.
    cbg = new CheckboxGroup();
    // Создать флажки и включить их в группу.
    windows = new Checkbox("Windows", cbg, true);
    android = new Checkbox("Android", cbg, false);
    linux = new Checkbox("Linux", cbg, false);
    mac = new Checkbox("Mac OS", cbg, false);
    // Добавить флажки во фрейм.
    add(windows);
    add(android);
    add(linux);
    add(mac);
    // Добавить прослушиватели событий элементов.
    windows.addItemListener(this);
    android.addItemListener(this);
    linux.addItemListener(this);
    mac.addItemListener(this);
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }
  // Отобразить текущее состояние флажков.
  public void paint(Graphics g) {
    msg = "Current selection: ";
        // Текущий выбор
    msg += cbg.getSelectedCheckbox().getLabel();
    g.drawString(msg, 20, 120);
  }
  public static void main(String[] args) {
    CBGroup appwin = new CBGroup();
    appwin.setSize(new Dimension(240, 180));
    appwin.setTitle("CBGroup");
    appwin.setVisible(true);
  }
}
```

Пример вывода из программы CBGroup показан на рис. 27.4. Обратите внимание, что флажки теперь имеют круглую форму.

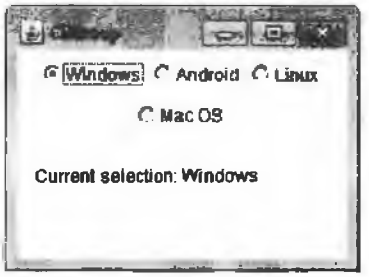


Рис. 27.4. Пример вывода из программы CBGroup

Элементы управления выбором

Класс Choice применяется для создания *раскрывающегося списка* элементов, среди которых пользователь может производить выбор. Таким образом, элемент управления Choice является разновидностью меню. Когда компонент Choice не активен, он занимает ровно столько места, сколько нужно для отображения текущего выбранного элемента. После щелчка пользователем на нем появляется полный список вариантов, где может быть сделан новый выбор. Каждый элемент в списке представляет собой строку, которая отображается в виде метки, выровненной по левому краю, в том порядке, в каком она добавлялась к объекту Choice. В классе Choice определен только стандартный конструктор, создающий пустой список.

Для добавления выбираемого элемента в список необходимо вызвать метод add():

```
void add(String name)
```

В name указывается имя добавляемого элемента. Элементы добавляются в список в порядке вызова метода add().

Чтобы определить, какой элемент выбран в текущий момент, можно вызвать либо getSelectedItem(), либо getSelectedIndex():

```
String getSelectedItem()
int getSelectedIndex()
```

Метод getSelectedItem() возвращает строку, содержащую имя элемента, а метод getSelectedIndex() — индекс элемента. Первый элемент имеет индекс 0. По умолчанию выбирается первый элемент, добавленный в список.

Чтобы получить количество элементов в списке, нужно вызвать метод getItemCount(). Для установки текущего выбранного элемента используется метод select(), которому передается либо целочисленный индекс, начинающийся с нуля, либо строка, соответствующая имени в списке. Указанные методы выглядят следующим образом:

```
int getItemCount()
void select(int index)
void select(String name)
```

При наличии индекса можно получить имя, ассоциированное с элементом по данному индексу, с помощью метода `getItem()`:

```
String getItem(int index)
```

В `index` указывается индекс интересующего элемента.

Обработка событий для списков выбора

Каждый раз, когда осуществляется выбор, генерируется событие элемента, которое отправляется всем прослушивателям, ранее зарегистрированным для получения уведомлений о событиях элемента от этого компонента. Все прослушиватели реализуют интерфейс `ItemListener`, в котором определен метод `itemStateChanged()`. В качестве аргумента методу `itemStateChanged()` передается объект `ItemEvent`.

Ниже показан пример создания двух меню `Choice`. Одно позволяет выбрать операционную систему, а другое — браузер.

```
// Демонстрация использования меню типа Choice.
import java.awt.*;
import java.awt.event.*;
public class ChoiceDemo extends Frame implements ItemListener {
  Choice os, browser;
  String msg = "";
  public ChoiceDemo() {
    // Использовать поточную компоновку.
    setLayout(new FlowLayout());

    // Создать списки выбора.
    os = new Choice();
    browser = new Choice();

    // Добавить элементы в список os.
    os.add("Windows");
    os.add("Android");
    os.add("Linux");
    os.add("Mac OS");

    // Добавить элементы в список browser.
    browser.add("Edge");
    browser.add("Firefox");
    browser.add("Chrome");

    // Добавить списки выбора во фрейм.
    add(os);
    add(browser);

    // Добавить прослушиватели событий элементов.
    os.addItemListener(this);
    browser.addItemListener(this);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
```

```
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }
  // Отобразить элементы, выбранные в текущий момент.
  public void paint(Graphics g) {
    msg = "Current OS: ";
        // Текущая операционная система
    msg += os.getSelectedItem();
    g.drawString(msg, 20, 120);
    msg = "Current Browser: ";
        // Текущий браузер
    msg += browser.getSelectedItem();
    g.drawString(msg, 20, 140);
  }
  public static void main(String[] args) {
    ChoiceDemo appwin = new ChoiceDemo();
    appwin.setSize(new Dimension(240, 180));
    appwin.setTitle("ChoiceDemo");
    appwin.setVisible(true);
  }
}
```

Пример вывода из программы ChoiceDemo показан на рис. 27.5.

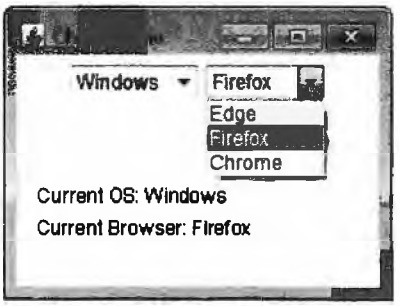


Рис. 27.5. Пример вывода из программы ChoiceDemo

Использование списков

Класс List предоставляет компактный прокручиваемый список с множественным выбором. В отличие от объекта Choice, который показывает только один выбранный пункт в меню, объект List способен отображать любое количество вариантов выбора в видимом окне. Его также можно создать так, чтобы сделать возможным множественный выбор. В классе List определены следующие конструкторы:

```
List() throws HeadlessException
List(int numRows) throws HeadlessException
List(int numRows, boolean multipleSelect) throws HeadlessException
```

Первая форма конструктора создает элемент управления List, который позволяет выбирать только один элемент в любой момент времени. Во вто-

рой форме значение numRows указывает количество записей в списке, которые будут всегда видны (другие можно прокручивать по мере необходимости). В третьей форме конструктора, если MultipleSelect имеет значение true, то пользователь может выбрать два или более элементов одновременно. Если оно равно false, тогда может быть выбран только один элемент.

Для добавления варианта выбора в список понадобится вызвать метод add(), имеющий две формы:

```
void add(String name)
void add(String name, int index)
```

В name указывается имя элемента, добавляемого в список. Первая форма метода добавляет элементы в конец списка. Вторая форма добавляет элемент по индексу, заданному в index. Индексация начинается с нуля. Чтобы добавить элемент в конец списка, необходимо указать -1.

Выяснить, какой элемент выбран в текущий момент, в списках, допускающих только одиночный выбор, можно с помощью метода getSelectedItem() или getSelectedIndex():

```
String getSelectedItem()
int getSelectedIndex()
```

Метод getSelectedItem() возвращает строку, содержащую имя элемента. Если выбрано более одного элемента или выбор еще не сделан, тогда возвращается null. Метод getSelectedIndex() возвращает индекс элемента. Первый элемент имеет индекс 0. Если выбрано более одного элемента или выбор еще не сделан, то возвращается -1.

Для определения текущего выбора в списках, допускающих множественный выбор, должен применяться метод getSelectedItems() или getSelectedIndexes():

```
String[] getSelectedItems()
int[] getSelectedIndexes()
```

Метод getSelectedItems() возвращает массив, содержащий имена выбранных в текущий момент элементов. Метод getSelectedIndexes() возвращает массив с индексами выбранных в данный момент элементов.

Для получения количества элементов в списке нужно вызвать метод getItemCount(). Установить текущий выбранный элемент можно посредством вызова метода select() с передачей ему целочисленного индекса, начинающегося с нуля. Вот упомянутые методы:

```
int getItemCount()
void select(int index)
```

Имея индекс, можно получить имя, ассоциированное с элементом по этому индексу, с помощью метода getItem(), который имеет следующую общую форму:

```
String getItem(int index)
```

В index указывается индекс желаемого элемента.

Обработка событий для списков

Для обработки событий списка потребуется реализовать интерфейс ActionListener. Каждый раз, когда происходит двойной щелчок на элементе списка, генерируется объект ActionEvent, метод getActionCommand() которого можно использовать для получения имени нового выбранного элемента. Кроме того, когда элемент выбирается или его выбор отменяется посредством одиночного щелчка, генерируется объект ItemEvent, метод getStateChange() которого можно применять с целью выяснения, что вызвало это событие — выбор или отмена выбора. Метод getItemSelectable() возвращает ссылку на объект, инициировавший событие.

В показанном далее примере элементы управления Choice из предыдущего раздела преобразуются в компоненты List — один с множественным выбором и другой с одиночным выбором:

```
// Демонстрация использования списков.

import java.awt.*;
import java.awt.event.*;

public class ListDemo extends Frame implements ActionListener {
  List os, browser;
  String msg = "";

  public ListDemo() {

    // Использовать поточную компоновку.
    setLayout(new FlowLayout());

    // Создать список с множественным выбором.
    os = new List(4, true);

    // Создать список с одиночным выбором.
    browser = new List(4);

    // Добавить элементы в список os.
    os.add("Windows");
    os.add("Android");
    os.add("Linux");
    os.add("Mac OS");

    // Добавить элементы в список browser.
    browser.add("Edge");
    browser.add("Firefox");
    browser.add("Chrome");

    // Установить начальный выбор.
    browser.select(1);
    os.select(0);

    // Добавить списки во фрейм.
    add(os);
    add(browser);

    // Добавить прослушиватели событий действий.
    os.addActionListener(this);
    browser.addActionListener(this);

    addWindowListener(new WindowAdapter() {
```

```
    public void windowClosing(WindowEvent we) {
      System.exit(0);
    }
  });
}
public void actionPerformed(ActionEvent ae) {
  repaint();
}
// Отобразить элементы, выбранные в текущий момент.
public void paint(Graphics g) {
  int[] idx;

  msg = "Current OS: ";
     // Текущая операционная система
  idx = os.getSelectedIndexes();
  for(int i=0; i<idx.length; i++)
    msg += os.getItem(idx[i]) + "   ";
  g.drawString(msg, 20, 120);
  msg = "Current Browser: ";
     // Текущий браузер
  msg += browser.getSelectedItem();
  g.drawString(msg, 20, 140);
}
public static void main(String[] args) {
  ListDemo appwin = new ListDemo();

  appwin.setSize(new Dimension(300, 180));
  appwin.setTitle("ListDemo");
  appwin.setVisible(true);
}
}
```

Пример вывода из программы `ListDemo` показан на рис. 27.6.

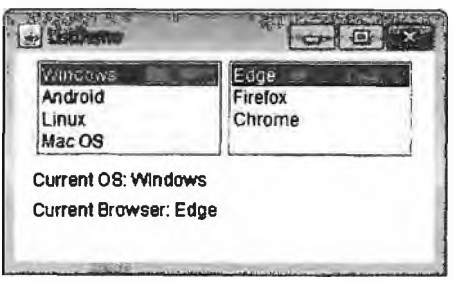


Рис. 27.6. Пример вывода из программы `ListDemo`

Управление полосами прокрутки

Полосы прокрутки используются для выбора непрерывных значений между заданными величинами минимума и максимума. Полосы прокрутки могут быть ориентированы горизонтально или вертикально. В действительности полоса прокрутки состоит из нескольких отдельных частей. На обоих кон-

цах расположены кнопки со стрелками, на которых можно щелкнуть, чтобы изменить текущее значение полосы прокрутки на одну единицу в направлении стрелки. Текущее значение полосы прокрутки относительно ее минимального и максимального значений указывает *ползунок* полосы прокрутки. Пользователь может перетащить ползунок в новое местоположение, и полоса прокрутки отразит соответствующее значение. Пользователь может щелкнуть на самой полосе по обе стороны от ползунка, чтобы переместить его в этом направлении с шагом, превышающим 1. Обычно такое действие транслируется в перелистывание на страницу вверх или вниз. Полосы прокрутки инкапсулированы в классе `Scrollbar`.

В классе `Scrollbar` определены следующие конструкторы:

```
Scrollbar() throws HeadlessException
Scrollbar(int style) throws HeadlessException
Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
  throws HeadlessException
```

Первая форма конструктора создает вертикальную полосу прокрутки. Вторая и третья формы позволяют указывать ориентацию полосы прокрутки. При передаче в `style` значения `Scrollbar.VERTICAL` создается вертикальная полоса прокрутки, а при передаче `Scrollbar.HORIZONTAL` — горизонтальная полоса прокрутки. В третьей форме конструктора параметр `initialValue` задает начальное значение для полосы прокрутки. В `thumbSize` указывается количество единиц, представляемых высотой ползунка. В параметрах `min` и `max` передаются минимальное и максимальное значения для полосы прокрутки.

В случае создания полосы прокрутки с применением одной из первых двух форм конструкторов необходимо установить ее параметры с помощью `setValues()`, прежде чем ее можно будет использовать:

```
void setValues(int initialValue, int thumbSize, int min, int max)
```

Параметры имеют такой же смысл, как в описанной выше третьей форме конструктора.

Чтобы получить текущее значение полосы прокрутки, нужно вызвать метод `getValue()`, а чтобы установить текущее значение, необходимо вызвать метод `setValue()`:

```
int getValue()
void setValue(int newValue)
```

В `newValue` указывается новое значение для полосы прокрутки. В результате установки значения ползунок внутри полосы прокрутки позиционируется в соответствии с новым значением.

Кроме того, с применением методов `getMinimum()` и `getMaximum()` можно получить минимальное и максимальное значения:

```
int getMinimum()
int getMaximum()
```

Они возвращают запрошенную величину.

По умолчанию прокрутка вверх или вниз на одну строку осуществляется с шагом 1. Шаг можно изменить, вызвав метод `setUnitIncrement()`. По умолчанию прокрутка вверх или вниз на одну страницу производится с шагом 10. Для его изменения понадобится вызвать метод `setBlockIncrement()`. Вот упомянутые методы:

```
void setUnitIncrement(int newIncr)
void setBlockIncrement(int newIncr)
```

Обработка событий для полос прокрутки

Для обработки событий полосы прокрутки потребуется реализовать интерфейс `AdjustmentListener`. Каждый раз, когда пользователь взаимодействует с полосой прокрутки, генерируется объект `AdjustmentEvent`, метод `getAdjustmentType()` которого можно использовать для определения вида корректировки. Виды событий корректировки кратко описаны в табл. 27.1.

Таблица 27.1. Виды событий корректировки

Событие	Описание
BLOCK_DECREMENT	Листание на страницу вниз
BLOCK_INCREMENT	Листание на страницу вверх
TRACK	Абсолютное отслеживание
UNIT_DECREMENT	Щелчок на кнопке перехода на строку вниз
UNIT_INCREMENT	Щелчок на кнопке перехода на строку вверх

В приведенном ниже примере создаются вертикальная и горизонтальная полосы прокрутки и отображаются их текущие настройки. При перетаскивании указателя мыши внутри окна координаты каждого события перетаскивания применяются для обновления полос прокрутки. В текущей позиции перетаскивания отображается символ звездочки. Обратите внимание на использование метода `setPreferredSize()` для установки размеров полос прокрутки.

```
// Демонстрация использования полос прокрутки.

import java.awt.*;
import java.awt.event.*;

public class SBDemo extends Frame implements AdjustmentListener {
  String msg = "";
  Scrollbar vertSB, horzSB;

  public SBDemo() {

    // Использовать поточную компоновку.
    setLayout(new FlowLayout());

    // Создать полосы прокрутки и установить предпочитаемые размеры.
    vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, 200);
    vertSB.setPreferredSize(new Dimension(20, 100));
```

```
    horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
                           0, 1, 0, 100);
    horzSB.setPreferredSize(new Dimension(100, 20));

    // Добавить полосы прокрутки во фрейм.
    add(vertSB);
    add(horzSB);

    // Добавить прослушиватели событий корректировки для полос прокрутки.
    vertSB.addAdjustmentListener(this);
    horzSB.addAdjustmentListener(this);

    // Добавить прослушиватели событий движения мыши.
    addMouseMotionListener(new MouseAdapter() {
      // Обновить полосы прокрутки, чтобы отразить
      // перетаскивание указателя мыши.
      public void mouseDragged(MouseEvent me) {
        int x = me.getX();
        int y = me.getY();
        vertSB.setValue(y);
        horzSB.setValue(x);
        repaint();
      }
    });

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void adjustmentValueChanged(AdjustmentEvent ae) {
    repaint();
  }

  // Отобразить текущие значения полос прокрутки.
  public void paint(Graphics g) {
    msg = "Vertical: " + vertSB.getValue();
    msg += ",  Horizontal: " + horzSB.getValue();
    g.drawString(msg, 20, 160);

    // Отобразить текущую позицию указателя мыши при перетаскивании.
    g.drawString("*", horzSB.getValue(),
                 vertSB.getValue());
  }

  public static void main(String[] args) {
    SBDemo appwin = new SBDemo();

    appwin.setSize(new Dimension(300, 180));
    appwin.setTitle("SBDemo");
    appwin.setVisible(true);
  }
}
```

Пример вывода из программы SBDemo показан на рис. 27.7.

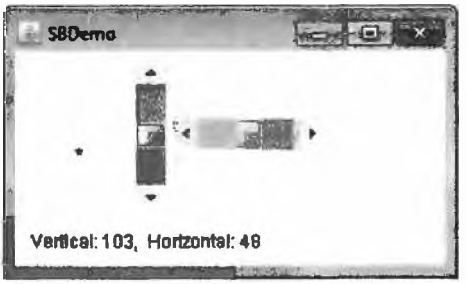


Рис. 27.7. Пример вывода из программы SBDemo

Использование текстовых полей

Класс TextField реализует однострочную область ввода текста, обычно называемую *элементом управления редактированием*. Текстовые поля позволяют пользователю вводить строки и редактировать текст с применением клавиш со стрелками, клавиш вырезания и вставки, а также выделения с помощью мыши. Класс TextField является подклассом TextComponent. В классе TextField определены следующие конструкторы:

```
TextField() throws HeadlessException
TextField(int numChars) throws HeadlessException
TextField(String str) throws HeadlessException
TextField(String str, int numChars) throws HeadlessException
```

Первая форма конструктора создает стандартное текстовое поле. Вторая форма конструирует текстовое поле с шириной numChars символов. Третья форма инициализирует текстовое поле строкой, содержащейся в str. Четвертая форма конструктора инициализирует текстовое поле и устанавливает его ширину.

Класс TextField и его суперкласс TextComponent предоставляют методы, позволяющие работать с текстовым полем. Для получения строки, которая в текущий момент содержится в текстовом поле, понадобится вызвать метод getText(), а для установки текста — метод setText():

```
String getText()
void setText(String str)
```

В str указывается новая строка.

Пользователь может выделять фрагмент текста в текстовом поле. Кроме того, фрагмент текста можно выбирать программно с использованием метода select(). Выделенный в текущий момент текст получается с помощью метода getSelectedText(). Вот эти методы:

```
String getSelectedText()
void select(int startIndex, int endIndex)
```

Метод getSelectedText() возвращает выделенный текст. Метод select() выбирает символы, начинающиеся с позиции startIndex и заканчивающиеся позицией endIndex-1.

Метод setEditable() позволяет управлять тем, может ли пользователь модифицировать содержимое текстового поля. Возможность редактирования выясняется с помощью метода isEditable(). Указанные методы имеют следующую форму:

```
boolean isEditable()
void setEditable(boolean canEdit)
```

Метод isEditable() возвращает true, если текст можно изменить, и false, если нет. Передав методу setEditable() в параметре canEdit значение true, можно разрешить изменение текста, тогда как значение false приведет к тому, что текст нельзя будет изменять.

Бывают случаи, когда нужно, чтобы вводимый пользователем текст, такой как пароль, не отображался. Отключить отображение символов по мере их ввода можно с применением метода setEchoChar(), который позволяет указать одиночный символ, отображаемый при наборе (таким образом, фактически введенные символы отображаться не будут). Метод echoCharIsSet() позволяет выяснить, находится ли текстовое поле в таком режиме. Получить отображаемый символ можно, вызвав метод getEchoChar(). Упомянутые методы показаны ниже:

```
void setEchoChar(char ch)
boolean echoCharIsSet()
char getEchoChar()
```

В ch указывается символ, который будет отображаться. При нулевом значении ch восстанавливается нормальное отображение вводимых символов.

Обработка событий для текстовых полей

Поскольку текстовые поля выполняют собственные функции редактирования, программа обычно не будет реагировать на события клавиатуры, возникающие в текстовом поле. Тем не менее, может понадобиться отреагировать на нажатие пользователем клавиши <Enter>, что приводит к генерации события действия.

В следующем примере создается классическая экранная форма для ввода имени пользователя и пароля:

```
// Демонстрация использования текстовых полей.

import java.awt.*;
import java.awt.event.*;

public class TextFieldDemo extends Frame implements ActionListener {
  TextField name, pass;
  public TextFieldDemo() {

    // Использовать поточную компоновку.
    setLayout(new FlowLayout());

    // Создать элементы управления.
    Label namep = new Label("Name: ", Label.RIGHT);          // Имя
    Label passp = new Label("Password: ", Label.RIGHT);      // Пароль
    name = new TextField(12);
```

```
    pass = new TextField(8);
    pass.setEchoChar('?');

    // Добавить элементы управления во фрейм.
    add(namep);
    add(name);
    add(passp);
    add(pass);

    // Добавить прослушиватели событий действий.
    name.addActionListener(this);
    pass.addActionListener(this);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  // Пользователь нажал клавишу <Enter>.
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  public void paint(Graphics g) {
    g.drawString("Name: " + name.getText(), 20, 100);    // Имя
    g.drawString("Selected text in name: "
                 + name.getSelectedText(), 20, 120); // Текст,
                                                     // выбранный в имени
    g.drawString("Password: " + pass.getText(), 20, 140);  // Пароль
  }

  public static void main(String[] args) {
    TextFieldDemo appwin = new TextFieldDemo();
    appwin.setSize(new Dimension(380, 180));
    appwin.setTitle("TextFieldDemo");
    appwin.setVisible(true);
  }
}
```

Пример вывода из программы `TextFieldDemo` показан на рис. 27.8. (Естественно, в реальном приложении должны быть решены проблемы безопасности, касающиеся паролей. Актуальные сведения о безопасности ищите в документации по Java.)

Рис. 27.8. Пример вывода из программы `TextFieldDemo`

Использование текстовых областей

Иногда одной строки для ввода текста недостаточно, чтобы решить имеющуюся задачу. Для таких ситуаций в библиотеке AWT предусмотрен простой многострочный редактор, реализуемый классом `TextArea` с показанными ниже конструкторами:

```
TextArea() throws HeadlessException
TextArea(int numLines, int numChars) throws HeadlessException
TextArea(String str) throws HeadlessException
TextArea(String str, int numLines, int numChars) throws HeadlessException
TextArea(String str, int numLines, int numChars, int sBars)
  throws HeadlessException
```

В `numLines` передается высота в строках текстовой области, а в `numChars` — ее ширина в символах. В `str` можно задать первоначальный текст. Пятая форма конструктора позволяет указать в параметре `sBars` полосы прокрутки, которые должен иметь элемент управления, посредством одного из следующих значений:

`SCROLLBARS_BOTH`	`SCROLLBARS_NONE`
`SCROLLBARS_HORIZONTAL_ONLY`	`SCROLLBARS_VERTICAL_ONLY`

Класс `TextArea` является подклассом `TextComponent`, поэтому он поддерживает методы `getText()`, `setText()`, `getSelectedText()`, `select()`, `isEditable()` и `setEditable()`, описанные в предыдущем разделе.

В классе `TextArea` добавляются новые методы для редактирования:

```
void append(String str)
void insert(String str, int index)
void replaceRange(String str, int startIndex, int endIndex)
```

Метод `append()` добавляет строку, указанную в `str`, в конец текущего текста. Метод `insert()` вставляет строку, заданную в `str`, по указанному индексу. Метод `replaceRange()` заменяет символы с индекса `startIndex` по индекс `endIndex-1` текстом замены, переданным в `str`.

Текстовые области являются почти автономными элементами управления. Накладные расходы, связанные с управлением, в программе практически отсутствуют. Обычно при необходимости просто получается текущий текст. Однако по желанию можно прослушивать события `TextEvent`.

В показанном далее примере создается элемент управления `TextArea`:

```
// Демонстрация использования TextArea.

import java.awt.*;
import java.awt.event.*;

public class TextAreaDemo extends Frame {
  public TextAreaDemo() {
    // Использовать поточную компоновку.
    setLayout(new FlowLayout());

    String val =
      "JDK 17 is the latest version of one of the most\n" +
```

```
      "widely-used computer languages for Internet programming.\n" +
      "Building on a rich heritage, Java has advanced both\n" +
      "the art and science of computer language design.\n\n" +
      "One of the reasons for Java's ongoing success is its\n" +
      "constant, steady rate of evolution. Java has never stood\n" +
      "still. Instead, Java has consistently adapted to the\n" +
      "rapidly changing landscape of the networked world.\n" +
      "Moreover, Java has often led the way, charting the\n" +
      "course for others to follow.";
    // JDK 17 -- последняя версия комплекта разработчика для одного
    // из наиболее широко используемых языков программирования
    // в Интернете.
    // Основываясь на богатом наследии, язык Java содействует развитию
    // как искусства, так и науки проектирования компьютерных языков.
    // Одной из причин постоянного успеха языка Java является его
    // постоянная и устойчивая скорость эволюции.
    // Язык Java никогда не стоял на месте, взамен последовательно
    // адаптируясь к быстро меняющемуся ландшафту сетевого мира.
    // Более того, язык Java часто шел впереди, намечая курс, которому
    // должны были следовать другие языки.
    TextArea text = new TextArea(val, 10, 30);
    add(text);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public static void main(String[] args) {
    TextAreaDemo appwin = new TextAreaDemo();

    appwin.setSize(new Dimension(300, 220));
    appwin.setTitle("TextAreaDemo");
    appwin.setVisible(true);
  }
}
```

Пример вывода из программы `TextAreaDemo` показан на рис. 27.9.

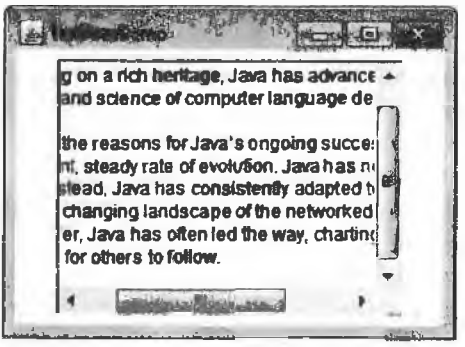


Рис. 27.9. Пример вывода из программы `TextAreaDemo`

Понятие диспетчеров компоновки

Все представленные до сих пор компоненты позиционировались с помощью диспетчера поточной компоновки (`FlowLayout`). Как упоминалось в начале главы, диспетчер компоновки автоматически размещает элементы управления в окне согласно определенному алгоритму. Хотя элементы управления Java можно размещать вручную, обычно так не поступают по двум главным причинам. Во-первых, позиционировать большое количество компонентов вручную крайне утомительно. Во-вторых, иногда при размещении какого-то элемента управления информация о ширине и высоте еще недоступна, поскольку инструментальные компоненты платформы пока не готовы. В такой ситуации довольно сложно понять, когда размеры заданного компонента можно применять для его позиционирования относительно другого.

С каждым объектом `Container` связан диспетчер компоновки, который представляет собой экземпляр любого класса, реализующего интерфейс `LayoutManager`. Диспетчер компоновки задается посредством метода `setLayout()`. Если вызов `setLayout()` не делался, тогда используется стандартный диспетчер компоновки. При каждом изменении размеров контейнера (либо при их первоначальной установке) диспетчер компоновки применяется для позиционирования всех компонентов внутри него.

Вот основная форма метода `setLayout()`:

```
void setLayout(LayoutManager layoutObj)
```

В `layoutObj` передается ссылка на нужный диспетчер компоновки. Если диспетчер компоновки необходимо отключить и позиционировать компоненты вручную, тогда в `layoutObj` понадобится передать `null`. В таком случае придется самостоятельно определять форму и местоположение каждого компонента с использованием метода `setBounds()`, определенного в `Component`. Обычно удобнее применять диспетчер компоновки.

Все диспетчеры компоновки отслеживает список компонентов, которые хранятся по своим именам. Диспетчер компоновки уведомляется при каждом добавлении компонента в контейнер. Всякий раз, когда размер контейнера нужно изменить, к диспетчеру компоновки обращаются через его методы `minimumLayoutSize()` и `preferredLayoutSize()`. Каждый компонент, управляемый диспетчером компоновки, содержит методы `getPreferredSize()` и `getMinimumSize()`, которые возвращают предпочтительный и минимальный размеры, необходимые для отображения каждого компонента. Диспетчер компоновки по возможности будет удовлетворять такие запросы, сохраняя целостность политики компоновки. Упомянутые методы можно переопределять в подклассах, создаваемых для элементов управления, иначе будут предоставляться стандартные значения.

В Java есть несколько предопределенных классов `LayoutManager`; часть из них описаны ниже. Можно использовать тот диспетчер компоновки, который лучше всего подходит для разрабатываемого приложения.

FlowLayout

Диспетчер компоновки FlowLayout уже демонстрировался в предшествующих примерах. Класс FlowLayout реализует простой стиль компоновки, похожий на размещение слов в текстовом редакторе. Направление компоновки определяется свойством ориентации компонента контейнера, которое по умолчанию устанавливает направление слева направо, сверху вниз. Таким образом, компоненты размещаются построчно, начиная с левого верхнего угла. Во всех случаях при заполнении строки производится переход на следующую строку. Между компонентами сверху и снизу, а также слева и справа оставляется небольшое пространство. Ниже приведены конструкторы класса FlowLayout:

```
FlowLayout()
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)
```

Первая форма конструктора создает стандартную компоновку, которая размещает компоненты по центру и оставляет между ними по пять пикселей пространства. Вторая форма позволяет указать, как выравнивается каждая строка. Вот допустимые значения параметра how:

```
FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
FlowLayout.LEADING
FlowLayout.TRAILING
```

Перечисленные значения задают выравнивание по левому краю, центру, правому краю, переднему краю и заднему краю соответственно. Третья форма конструктора дает возможность указать в horz и vert горизонтальный и вертикальный промежутки между компонентами. Результат указания выравнивания с помощью FlowLayout можно увидеть, заменив строку в показанной ранее программе CheckboxDemo следующим образом:

```
setLayout(new FlowLayout(FlowLayout.LEFT));
```

После внесения такого изменения вывод будет выглядеть, как показано на рис. 27.10. Сравните его с выводом из исходной версии программы, который был представлен на рис. 27.3.

Рис. 27.10. Пример вывода из программы CheckboxDemo после внесения изменения

BorderLayout

Класс BorderLayout реализует стиль компоновки с четырьмя узкими компонентами фиксированной ширины по краям и одной большой областью в центре. Четыре стороны именуются по названиям сторон света: север, юг, восток и запад. Область посредине называется центром. Класс BorderLayout является стандартным диспетчером компоновки для Frame. Ниже представлены конструкторы, определенные в BorderLayout:

```
BorderLayout()
BorderLayout(int horz, int vert)
```

Первая форма конструктора создает стандартную граничную компоновку. Вторая форма позволяет указать в horz и vert горизонтальный и вертикальный промежутки между компонентами.

В BorderLayout определены следующие часто применяемые константы для указания областей:

```
BorderLayout.CENTER          BorderLayout.SOUTH
BorderLayout.EAST            BorderLayout.WEST
BorderLayout.NORTH
```

Перечисленные константы будут использоваться при добавлении компонентов с помощью приведенного далее метода add(), определенного в классе Container:

```
void add(Component compRef, Object region)
```

В compRef передается ссылка на добавляемый компонент, а в region указывается, куда будет добавлен компонент.

Вот пример BorderLayout с компонентом в каждой области компоновки:

```
// Демонстрация использования BorderLayout.

import java.awt.*;
import java.awt.event.*;

public class BorderLayoutDemo extends Frame {
  public BorderLayoutDemo() {
    // В данном случае BorderLayout применяется по умолчанию.
    add(new Button("This is across the top."), BorderLayout.NORTH);
                  // Это располагается вверху.
    add(new Label("The footer message might go here."), BorderLayout.SOUTH);
                // Здесь может находиться нижний колонтитул.
    add(new Button("Right"), BorderLayout.EAST);
                  // Справа
    add(new Button("Left"), BorderLayout.WEST);
                  // Слева

    String msg = "The reasonable man adapts " +
                 "himself to the world,\n" +
                 "the unreasonable one persists in " +
                 "trying to adapt the world to himself.\n" +
                 "Therefore all progress depends " +
```

```
            "on the unreasonable man.\n\n" +
            "    - George Bernard Shaw\n\n";
          // Рациональный человек приспосабливается к миру,
          // безрассудный же упорно пытается приспособить мир к себе.
          // Поэтому весь прогресс зависит от безрассудных людей.
          //     - Джордж Бернард Шоу
    add(new TextArea(msg), BorderLayout.CENTER);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public static void main(String[] args) {
    BorderLayoutDemo appwin = new BorderLayoutDemo();

    appwin.setSize(new Dimension(300, 220));
    appwin.setTitle("BorderLayoutDemo");
    appwin.setVisible(true);
  }
}
```

Пример вывода из программы BorderLayoutDemo показан на рис. 27.11.

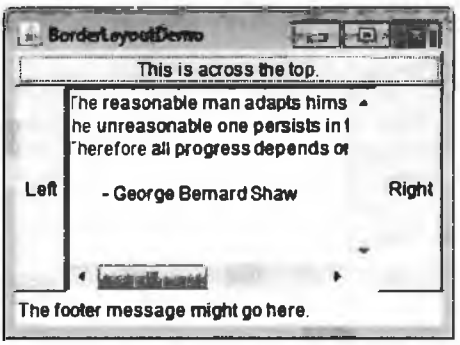


Рис. 27.11. Пример вывода из программы BorderLayoutDemo

Использование вставок

Иногда может требоваться оставлять небольшое пространство между контейнером, содержащим компоненты, и окном, в котором они находятся, для чего необходимо переопределить метод getInsets() класса Container. Данный метод возвращает объект типа Insets, содержащий верхнюю, нижнюю, левую и правую вставки, которые будут применяться при отображении контейнера. Такие значения используются диспетчером компоновки для вставки компонентов при компоновке окна. Вот конструктор класса Insets:

```
Insets(int top, int left, int bottom, int right)
```

Значения, передаваемые в top, left, bottom и right, задают расстояния между контейнером и окружающим его окном.

Метод `getInsets()` имеет следующую общую форму:

```
Insets getInsets()
```

В случае переопределения метода `getInsets()` понадобится возвратить новый объект `Insets`, который содержит желаемый промежуток для вставки.

Ниже показан предыдущий пример программы `BorderLayout`, модифицированный с целью вставки компонентов. В качестве фонового цвета выбран циан, чтобы вставки были лучше видны.

```
// Демонстрация применения BorderLayout со вставками.
import java.awt.*;
import java.awt.event.*;

public class InsetsDemo extends Frame {
  public InsetsDemo() {
    // В данном случае BorderLayout применяется по умолчанию.

    // Установить фоновый цвет, чтобы вставки было легко заметить.
    setBackground(Color.cyan);

    setLayout(new BorderLayout());

    add(new Button("This is across the top."), BorderLayout.NORTH);
    add(new Label("The footer message might go here."),BorderLayout.SOUTH);
    add(new Button("Right"), BorderLayout.EAST);
    add(new Button("Left"), BorderLayout.WEST);

    String msg = "The reasonable man adapts " +
                 "himself to the world;\n" +
                 "the unreasonable one persists in " +
                 "trying to adapt the world to himself.\n" +
                 "Therefore all progress depends " +
                 "on the unreasonable man.\n\n" +
                 "    - George Bernard Shaw\n\n";
    add(new TextArea(msg), BorderLayout.CENTER);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  // Переопределить метод getInsets(), чтобы добавить значения вставок.
  public Insets getInsets() {
    return new Insets(40, 20, 10, 20);
  }

  public static void main(String[] args) {
    InsetsDemo appwin = new InsetsDemo();

    appwin.setSize(new Dimension(300, 220));
    appwin.setTitle("InsetsDemo");
    appwin.setVisible(true);
  }
}
```

Пример вывода из программы `InsetsDemo` показан на рис. 27.12.

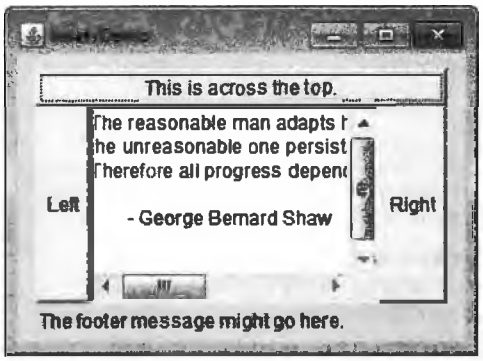


Рис. 27.12. Пример вывода из программы `InsetsDemo`

GridLayout

Класс `GridLayout` размещает компоненты в двумерной экранной сетке. При создании экземпляра `GridLayout` указывается количество строк и колонок. В классе `GridLayout` определены следующие конструкторы:

```
GridLayout()
GridLayout(int numRows, int numColumns)
GridLayout(int numRows, int numColumns, int horz, int vert)
```

Первая форма конструктора создает сеточную компоновку с одной колонкой, а вторая форма — сеточную компоновку с заданным количеством строк и колонок. Третья форма конструктора позволяет указывать в `horz` и `vert` промежутки по горизонтали и вертикали, оставляемые между компонентами. Значение `numRows` или `numColumns` может быть нулевым. Передача `0` для `numRows` позволяет применять колонки неограниченной длины, а передача `0` для `numColumns` дает возможность использовать строки неограниченной длины.

В показанной далее программе создается сетка 4×4, которая заполняется 15 кнопками с пометкой каждой кнопки своим индексом:

```
// Демонстрация использования GridLayout.

import java.awt.*;
import java.awt.event.*;

public class GridLayoutDemo extends Frame {
  static final int n = 4;

  public GridLayoutDemo() {

    // Использовать GridLayout.
    setLayout(new GridLayout(n, n));
    setFont(new Font("SansSerif", Font.BOLD, 24));
    for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
        int k = i * n + j;
        if(k > 0)
          add(new Button("" + k));
      }
    }
```

```
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public static void main(String[] args) {
    GridLayoutDemo appwin = new GridLayoutDemo();
    appwin.setSize(new Dimension(300, 220));
    appwin.setTitle("GridLayoutDemo");
    appwin.setVisible(true);
  }
}
```

Пример вывода из программы GridLayoutDemo показан на рис. 27.13.

Рис. 27.13. Пример вывода из программы GridLayoutDemo

Совет. Приведенный пример можно взять за основу для создания игры в 15.

CardLayout

Среди других диспетчеров компоновки класс CardLayout уникален тем, что в нем хранится несколько разных компоновок. Каждую компоновку можно рассматривать как отдельную карту в колоде, которую можно перетасовать так, чтобы любая карта оказалась сверху в данный момент времени. Такая возможность полезна для пользовательских интерфейсов с дополнительными компонентами, которые допускается динамически включать и отключать при вводе данных пользователем. Другие компоновки можно подготовить заблаговременно и скрыть их, чтобы затем при необходимости активизировать.

Вот конструкторы, предоставляемые классом CardLayout:

```
CardLayout()
CardLayout(int horz, int vert)
```

Первая форма конструктора создает стандартную карточную компоновку. Вторая форма позволяет указать в horz и vert промежутки по горизонтали и вертикали, оставляемые между компонентами.

Использование карточной компоновки требует чуть большей работы, чем в случае других компоновок. Карты обычно хранятся в объекте типа Panel, для которого в качестве диспетчера компоновки должен быть выбран CardLayout. Карты, образующие колоду, тоже обычно являются объектами типа Panel. Таким образом, понадобится создать панель, содержащую колоду, и панель для каждой карты в колоде. Затем к соответствующей панели добавляются компоненты, из которых состоит каждая карта. Далее готовые панели добавляются в панель с диспетчером компоновки CardLayout. Наконец, последняя панель добавляется в окно. После выполнения описанных шагов пользователю нужно предоставить возможность выбора между картами. Одним из распространенных подходов является добавление по одной кнопке для каждой карты в колоде.

Когда карточные панели добавляются в панель, им обычно назначаются имена, для чего применяется показанная ниже форма метода add():

```
void add(Component panelRef, Object name)
```

В name передается строка с именем карты, панель которой указана в panelRef.

После создания колоды можно активизировать какую-то карту, вызвав один из следующих методов класса CardLayout:

```
void first(Container deck)
void last(Container deck)
void next(Container deck)
void previous(Container deck)
void show(Container deck, String cardName)
```

В deck передается ссылка на контейнер (обычно панель), в котором хранятся карты, а в cardName — имя карты. Вызов метода first() приводит к отображению первой карты в колоде, а вызов last() — последней карты. Чтобы отобразить следующую карту, понадобится вызвать метод next(), а для отображения предыдущей карты — метод previous(). Методы next() и previous() автоматически переходят к верхней или нижней части колоды. Метод show() отображает карту, имя которой указывается в cardName.

В следующем примере создается двухуровневая колода карт, которая позволяет пользователю выбрать операционную систему. Внутри одной карты отображаются операционные системы на базе Windows, а внутри другой — операционные системы Mac OS, Android и Linux.

```
// Демонстрация использования CardLayout.

import java.awt.*;
import java.awt.event.*;

public class CardLayoutDemo extends Frame {
  Checkbox windows10, windows7, windows8, android, linux, mac;
  Panel osCards;
  CardLayout cardLO;
  Button win, other;
```

```
public CardLayoutDemo() {
  // Использовать поточную компоновку для главного фрейма.
  setLayout(new FlowLayout());
  win = new Button("Windows");
  other = new Button("Other");
  add(win);
  add(other);
  // Обеспечить применение в панели osCards компоновки CardLayout.
  cardLO = new CardLayout();
  osCards = new Panel();
  osCards.setLayout(cardLO);
  windows7 = new Checkbox("Windows 7", true);
  windows8 = new Checkbox("Windows 8");
  windows10 = new Checkbox("Windows 10");
  android = new Checkbox("Android");
  linux = new Checkbox("Linux");
  mac = new Checkbox("Mac OS");
  // Добавить в панель флажки для операционных систем семейства Windows.
  Panel winPan = new Panel();
  winPan.add(windows7);
  winPan.add(windows8);
  winPan.add(windows10);
  // Добавить в панель флажки для других операционных систем.
  Panel otherPan = new Panel();
  otherPan.add(android);
  otherPan.add(linux);
  otherPan.add(mac);
  // Добавить панели в панель колоды карт.
  osCards.add(winPan, "Windows");
  osCards.add(otherPan, "Other");
                        // Другие
  // Добавить карты в панель главного фрейма.
  add(osCards);
  // Использовать лямбда-выражения для обработки кнопочных событий.
  win.addActionListener((ae) -> cardLO.show(osCards, "Windows"));
  other.addActionListener((ae) -> cardLO.show(osCards, "Other"));
  // Зарегистрировать события нажатия кнопки мыши.
  addMouseListener(new MouseAdapter() {
    // Совершить проход по панелям.
    public void mousePressed(MouseEvent me) {
      cardLO.next(osCards);
    }
  });
  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent we) {
      System.exit(0);
    }
  });
}
```

```
  public static void main(String[] args) {
    CardLayoutDemo appwin = new CardLayoutDemo();
    appwin.setSize(new Dimension(300, 220));
    appwin.setTitle("CardLayoutDemo");
    appwin.setVisible(true);
  }
}
```

Пример вывода из программы CardLayoutDemo показан на рис. 27.14. Каждая карта активизируется нажатием кнопки, связанной с картой. Для прохода по картам можно также просто щелкать кнопкой мыши.

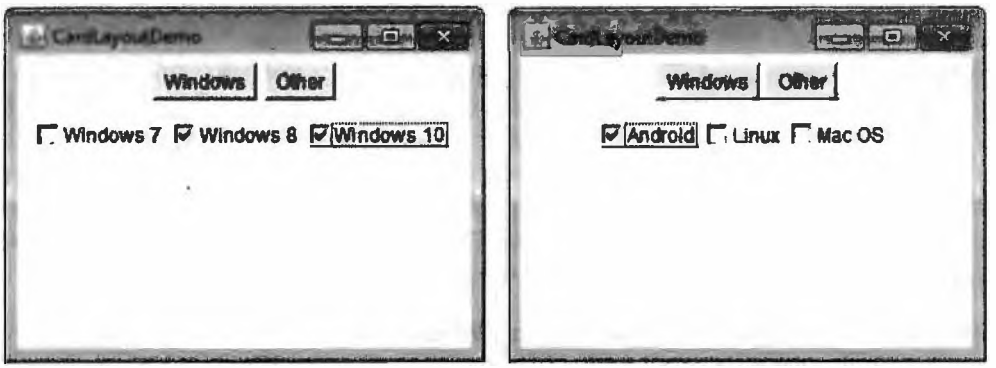


Рис. 27.14. Пример вывода из программы CardLayoutDemo

GridBagLayout

Хотя рассмотренные ранее компоновки идеально подходят для многих целей, в некоторых ситуациях требуется чуть больший контроль над тем, как располагаются компоненты. Эффективный способ предусматривает использование гибкой сеточной компоновки, обеспечиваемой классом GridBagLayout. Гибкая сеточная компоновка полезна тем, что позволяет задавать относительное расположение компонентов, указывая их позиции в ячейках внутри сетки. Основная ее особенность заключается в том, что каждый компонент может иметь разные размеры, а каждая строка в сетке способна содержать разное количество колонок. Именно потому такая компоновка называется *гибкой*. Она выглядит как множество соединенных вместе маленьких сеток.

Местоположение и размеры каждого компонента в гибкой сеточной компоновке определяются набором связанных с ним ограничений. Ограничения содержатся в объекте типа GridBagConstraints. В состав ограничений входит высота и ширина ячейки, а также размещение компонента, его выравнивание и точка привязки внутри ячейки.

Согласно общей процедуре работы с гибкой сеточной компоновкой сначала создается новый объект GridBagLayout и делается текущим диспетчером компоновки. Затем понадобится установить ограничения, которые применяются к каждому компоненту, который будет добавлен в гибкую сеточную

компоновку. Наконец, компоненты добавляются в диспетчер компоновки. Хотя GridBagLayout немного сложнее, чем другие диспетчеры компоновки, его все же довольно легко использовать, если вы понимаете, как иметь с ним дело.

В классе GridBagLayout определен единственный конструктор:

```
GridBagLayout()
```

Кроме того, в GridBagLayout определены методы, многие из которых являются защищенными и не предназначенными для широкого применения. Тем не менее, один из них использовать придется — setConstraints():

```
void setConstraints(Component comp, GridBagConstraints cons)
```

Параметр comp — это компонент, для которого применяются ограничения, указанные в cons. Метод setConstraints() устанавливает ограничения, которые применяются к каждому компоненту в гибкой сеточной компоновке.

Основополагающей предпосылкой для успешного использования GridBagLayout является надлежащая установка ограничений, хранящихся в объекте GridBagConstraints. В классе GridBagConstraints определено несколько полей, которые можно настроить для управления размером, размещением и расстоянием между компонентами. Все они кратко описаны в табл. 27.2, а некоторые обсуждаются более подробно далее в главе.

Таблица 27.2. Поля ограничений, определенные в классе GridBagConstraints

Поле	Описание
int anchor	Указывает местоположение компонента внутри ячейки. По умолчанию принимается GridBagConstraints.CENTER
int fill	Указывает, каким образом компонент изменяет размеры, если он меньше своей ячейки. Допустимые значения: GridBagConstraints.NONE (стандартное), GridBagConstraints.HORIZONTAL, GridBagConstraints.VERTICAL, GridBagConstraints.BOTH
int gridheight	Указывает высоту компонента в ячейках. По умолчанию принимается 1
int gridwidth	Указывает ширину компонента в ячейках. По умолчанию принимается 1
int gridx	Указывает координату X ячейки, в которую будет добавлен компонент. По умолчанию принимается GridBagConstraints.RELATIVE
int gridy	Указывает координату Y ячейки, в которую будет добавлен компонент. По умолчанию принимается GridBagConstraints.RELATIVE

Окончание табл. 27.2

Поле	Описание
`Insets insets`	Указывает вставки. По умолчанию все вставки принимаются равными 0
`int ipadx`	Указывает дополнительный промежуток по горизонтали, окружающий компонент внутри ячейки. По умолчанию принимается 0
`int ipady`	Указывает дополнительный промежуток по вертикали, окружающий компонент внутри ячейки. По умолчанию принимается 0
`double weightx`	Указывает весовое значение, которое определяет промежутки по горизонтали между ячейками и краями содержащего их контейнера. По умолчанию принимается 0.0. Чем выше весовое значение, тем больше пространства выделяется. Если все значения равны 0.0, тогда дополнительные промежутки равномерно распределяется между краями окна
`double weighty`	Указывает весовое значение, которое определяет промежутки по вертикали между ячейками и краями содержащего их контейнера. По умолчанию принимается 0.0. Чем выше весовое значение, тем больше пространства выделяется. Если все значения равны 0.0, тогда дополнительные промежутки равномерно распределяется между краями окна

В классе `GridBagConstraints` также определены статические поля, содержащие стандартные значения ограничений, например, `GridBagConstraints.CENTER` и `GridBagConstraints.VERTICAL`.

Когда компонент по размерам меньше своей ячейки, можно применять поле `anchor`, чтобы указать, где в ячейке будет располагаться верхний левый угол компонента. Существуют три типа значений, которые разрешено присваивать полю `anchor`. Первый тип — абсолютные значения:

```
GridBagConstraints.CENTER
GridBagConstraints.EAST
GridBagConstraints.NORTH
GridBagConstraints.NORTHEAST
GridBagConstraints.NORTHWEST
GridBagConstraints.SOUTH
GridBagConstraints.SOUTHEAST
GridBagConstraints.SOUTHWEST
GridBagConstraints.WEST
```

Как следует из имен значений, они заставляют компонент размещаться в определенных местах.

Значения второго типа, которые могут быть заданы в поле anchor, являются относительными к ориентации контейнера, которая может отличаться в случае языков, отличающихся от западных. Относительные значения перечислены ниже:

```
GridBagConstraints.FIRST_LINE_END       GridBagConstraints.LINE_END
GridBagConstraints.FIRST_LINE_START     GridBagConstraints.LINE_START
GridBagConstraints.LAST_LINE_END        GridBagConstraints.PAGE_END
GridBagConstraints.LAST_LINE_START      GridBagConstraints.PAGE_START
```

Их имена описывают расположение.

Значения третьего типа, которые можно указывать в поле anchor, позволяют позиционировать компоненты относительно базовой линии строки. Вот эти значения:

```
GridBagConstraints.BASELINE                GridBagConstraints.BASELINE_LEADING
GridBagConstraints.BASELINE_TRAILING       GridBagConstraints.ABOVE_BASELINE
GridBagConstraints.ABOVE_BASELINE_LEADING  GridBagConstraints.ABOVE_BASELINE_TRAILING

GridBagConstraints.BELOW_BASELINE          GridBagConstraints.BELOW_BASELINE_LEADING

GridBagConstraints.BELOW_BASELINE_TRAILING
```

Горизонтальная позиция может находиться по центру, по переднему краю (LEADING) или по заднему краю (TRAILING).

Поля weightx и weighty очень важны, хотя на первый взгляд сбивают с толку. Как правило, их значения определяют, сколько дополнительного пространства в контейнере выделяется для каждой строки и колонки. По умолчанию оба поля имеют нулевые значения. Когда все значения в строке или колонке равны нулю, дополнительное пространство равномерно распределяется между краями окна. За счет увеличения веса увеличивается выделение пространства данной строке или колонке пропорционально другим строкам или колонкам. Лучший способ понять, как работают такие значения — немного поэкспериментировать с ними.

Переменная gridwidth позволяет задать ширину ячейки в единицах ячейки. Стандартное значение равно 1. Для указания о том, что компонент использует оставшееся пространство в строке, нужно применить GridBagConstraints.REMAINDER. Чтобы указать, что компонент использует предпоследнюю ячейку в строке, необходимо применить GridBagConstraints.RELATIVE. Ограничение gridheight работает аналогично, но в вертикальном направлении.

Можно указать значение заполнения, которое будет использоваться для увеличения минимального размера ячейки. Для горизонтального заполнения понадобится присвоить значение ipadx, а для вертикального заполнения — ipady.

Ниже показан пример, в котором GridBagLayout применяется для демонстрации некоторых только что обсуждавшихся моментов:

```
// Демонстрация использования GridBagLayout.
import java.awt.*;
import java.awt.event.*;

public class GridBagDemo extends Frame implements ItemListener {
  String msg = "";
  Checkbox windows, android, linux, mac;
  public GridBagDemo() {
    // Использовать GridBagLayout.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);
    // Определить флажки.
    windows = new Checkbox("Windows ", true);
    android = new Checkbox("Android");
    linux = new Checkbox("Linux");
    mac = new Checkbox("Mac OS");
    // Определить гибкую сеточную компоновку.
    // Использовать стандартный вес строки, равный 0, для первой строки.
    gbc.weightx = 1.0;                    // вес колонки, равный 1
    gbc.ipadx = 200;                      // заполнение на 200 единиц
    gbc.insets = new Insets(0, 6, 0, 0);  // вставка немного слева
    gbc.anchor = GridBagConstraints.NORTHEAST;
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(windows, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(android, gbc);

    // Назначить второй строке вес, равный 1.
    gbc.weighty = 1.0;

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(linux, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(mac, gbc);

    // Добавить компоненты.
    add(windows);
    add(android);
    add(linux);
    add(mac);

    // Зарегистрировать прослушиватели для получения событий элементов.
    windows.addItemListener(this);
    android.addItemListener(this);
    linux.addItemListener(this);
    mac.addItemListener(this);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
```

```
  // Перерисовывать в случае изменения состояния какого-то флажка.
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }
  // Отобразить текущее состояние флажков.
  public void paint(Graphics g) {
    msg = "Current state: ";
        // Текущее состояние
    g.drawString(msg, 20, 100);
    msg = "  Windows: " + windows.getState();
    g.drawString(msg, 30, 120);
    msg = "  Android: " + android.getState();
    g.drawString(msg, 30, 140);
    msg = "   Linux: " + linux.getState();
    g.drawString(msg, 30, 160);
    msg = "   Mac OS: " + mac.getState();
    g.drawString(msg, 30, 180);
  }
  public static void main(String[] args) {
    GridBagDemo appwin = new GridBagDemo();
    appwin.setSize(new Dimension(250, 200));
    appwin.setTitle("GridBagDemo");
    appwin.setVisible(true);
  }
}
```

Пример вывода из программы `GridBagDemo` показан на рис. 27.15.

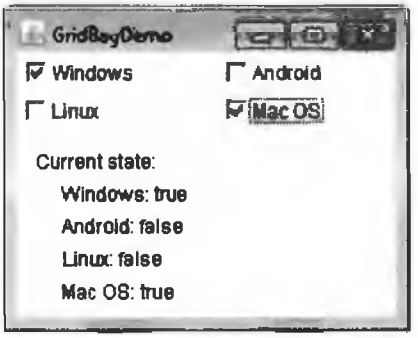


Рис. 27.15. Пример вывода из программы `GridBagDemo`

В созданной компоновке флажки для операционных систем расположены в сетке 2×2. Каждая ячейка имеет горизонтальное заполнение, равное 200. Каждый компонент слева немного смещен (на 6 единиц). Вес колонки установлен в 1, что приводит к равномерному распределению дополнительного пространства по горизонтали между колонками. В первой строке используется стандартный вес, равный 0, а во второй строке — вес 1. Таким образом, любое дополнительное пространство по вертикали добавляется ко второй строке.

Класс GridBagLayout является мощным диспетчером компоновки и потому стоит потратить некоторое время на его исследование посредством экспериментирования. Как только вы поймете, что делают различные настройки, то сможете применять GridBagLayout для позиционирования компонентов с высокой степенью точности.

Меню и панели меню

Окно верхнего уровня может иметь ассоциированную с ним панель меню, которая отображает список пунктов меню верхнего уровня. С каждым пунктом связано раскрывающееся меню. Такая концепция реализована в AWT классами MenuBar, Menu и MenuItem. Обычно панель меню содержит один или несколько объектов Menu, каждый из которых включает список объектов MenuItem. Каждый объект MenuItem представляет то, что может быть выбрано пользователем. Поскольку Menu является подклассом MenuItem, можно создать иерархию вложенных подменю. Кроме того, с помощью класса CheckboxMenuItem можно предусмотреть переключаемые пункты меню, при выборе которых рядом появляется галочка.

Для создания панели меню сначала понадобится создать экземпляр класса MenuBar, в котором определен только стандартный конструктор. Затем необходимо создать экземпляры Menu, которые будут определять выбор, отображаемый в панели меню. Ниже приведены конструкторы класса Menu:

```
Menu() throws HeadlessException
Menu(String optionName) throws HeadlessException
Menu(String optionName, boolean removable) throws HeadlessException
```

В optionName указывается имя пункта меню. Если в параметре removable передается значение true, тогда меню можно удалять и переводить в плавающий режим. В противном случае меню останется прикрепленным к панели меню. (Удаляемые меню зависят от реализации.) Первая форма конструктора создает пустое меню.

Отдельные пункты меню относятся к классу MenuItem, в котором определены следующие конструкторы:

```
MenuItem() throws HeadlessException
MenuItem(String itemName) throws HeadlessException
MenuItem(String itemName, MenuShortcut keyAccel) throws HeadlessException
```

В itemName передается имя, отображаемое в меню, а в keyAccel — клавиша быстрого доступа к данному пункту меню.

Пункт меню можно отключать и включить с использованием метода setEnabled():

```
void setEnabled(boolean enabledFlag)
```

Если в enableFlag передается значение true, тогда пункт меню включается, а если false, то отключается. Состояние пункта меню можно выяснить с применением метода isEnabled():

```
boolean isEnabled()
```

Метод isEnabled() возвращает true, если пункт меню, для которого он был вызван, включен, или false в противном случае.

Метод setLabel() позволяет изменить имя пункта меню, а метод getLabel() — получить текущее имя пункта меню:

```
void setLabel(String newName)
String getLabel()
```

Имя, указанное в newName, становится новым именем вызывающего пункта меню. Метод getLabel() возвращает текущее имя.

Создать переключаемый пункт меню можно с использованием подкласса MenuItem по имени CheckboxMenuItem, который имеет следующие конструкторы:

```
CheckboxMenuItem() throws HeadlessException
CheckboxMenuItem(String itemName) throws HeadlessException
CheckboxMenuItem(String itemName, boolean on) throws HeadlessException
```

В itemName передается имя пункта, отображаемого в меню. Переключаемый пункт меню работает как тумблер. Каждый раз, когда он выбирается, его состояние изменяется. В первых двух формах конструкторов переключаемый пункт меню не отмечен. В третьей форме, если on равно true, тогда переключаемый пункт меню изначально отмечен, а если false, то нет.

Получить состояние переключаемого пункта меню можно с помощью метода getState(), а установить его в известное состояние — посредством метода setState():

```
boolean getState()
void setState(boolean checked)
```

Если пункт меню отмечен, тогда getState() возвращает true, а если не отмечен, то false. Чтобы отметить пункт меню, нужно передать методу setState() значение true. Чтобы снять отметку с пункта меню, необходимо передать setState() значение false.

Созданный пункт меню потребуется добавить в объект Menu с применением метода add(), который имеет следующую общую форму:

```
MenuItem add(MenuItem item)
```

В item передается добавляемый пункт. Пункты добавляются в меню в порядке вызова метода add(). Метод возвращает item.

После добавления всех пунктов в объект Menu этот объект можно добавить в панель меню, используя версию add(), которая определена в классе MenuBar:

```
Menu add(Menu menu)
```

В menu передается добавляемое меню. Метод возвращает menu.

Меню генерируют события только при выборе элемента типа MenuItem или CheckboxMenuItem. В случае доступа к панели меню, скажем, для отображения раскрывающегося меню, события не генерируются. Каждый раз, когда выбирается пункт меню, создается объект ActionEvent. По умолча-

нию строкой с командой действия будет имя пункта меню, но можно указать другую строку с командой действия, вызвав метод `setActionCommand()` на пункте меню. При отметке или снятии отметки с переключаемого пункта меню создается объект `ItemEvent`. Таким образом, для обработки этих событий меню должны быть реализованы интерфейсы `ActionListener` и/или `ItemListener`.

Метод `getItem()` класса `ItemEvent` возвращает ссылку на элемент, сгенерировавший данное событие. Вот его общая форма:

```
Object getItem()
```

Ниже приведен пример добавления набора вложенных меню во всплывающее окно. В окне отображается выбранный пункт, а также состояние двух переключаемых пунктов меню.

```
// Демонстрация использования меню.

import java.awt.*;
import java.awt.event.*;

class MenuDemo extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;

  public MenuDemo() {

    // Создать панель меню и добавить ее во фрейм.
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // Создать пункты меню.
    Menu file = new Menu("File");                    // Файл
    MenuItem item1, item2, item3, item4, item5;
    file.add(item1 = new MenuItem("New..."));        // Создать
    file.add(item2 = new MenuItem("Open..."));       // Открыть
    file.add(item3 = new MenuItem("Close"));         // Закрыть
    file.add(item4 = new MenuItem("-"));
    file.add(item5 = new MenuItem("Quit..."));       // Выйти
    mbar.add(file);

    Menu edit = new Menu("Edit");                    // Правка
    MenuItem item6, item7, item8, item9;
    edit.add(item6 = new MenuItem("Cut"));           // Вырезать
    edit.add(item7 = new MenuItem("Copy"));          // Скопировать
    edit.add(item8 = new MenuItem("Paste"));         // Вставить
    edit.add(item9 = new MenuItem("-"));

    Menu sub = new Menu("Special");                  // Сервис
    MenuItem item10, item11, item12;
    sub.add(item10 = new MenuItem("First"));         // Первый
    sub.add(item11 = new MenuItem("Second"));        // Второй
    sub.add(item12 = new MenuItem("Third"));         // Третий
    edit.add(sub);

    // Создать переключаемые пункты меню.
    debug = new CheckboxMenuItem("Debug");           // Отладка
    edit.add(debug);
    test = new CheckboxMenuItem("Testing");          // Тестирование
```

```
    edit.add(test);

    mbar.add(edit);

    // Создать объект для обработки событий действий и элементов.
    MyMenuHandler handler = new MyMenuHandler();

    // Зарегистрировать для получения этих событий.
    item1.addActionListener(handler);
    item2.addActionListener(handler);
    item3.addActionListener(handler);
    item4.addActionListener(handler);
    item6.addActionListener(handler);
    item7.addActionListener(handler);
    item8.addActionListener(handler);
    item9.addActionListener(handler);
    item10.addActionListener(handler);
    item11.addActionListener(handler);
    item12.addActionListener(handler);
    debug.addItemListener(handler);
    test.addItemListener(handler);

    // Использовать лямбда-выражение для обработки выбора пункта Quit.
    item5.addActionListener((ae) -> System.exit(0));

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    g.drawString(msg, 10, 220);

    if(debug.getState())
      g.drawString("Debug is on.", 10, 240);    // Пункт Debug отмечен.
    else
      g.drawString("Debug is off.", 10, 240);   // Пункт Debug не отмечен.

    if(test.getState())
      g.drawString("Testing is on.", 10, 260);  // Пункт Testing отмечен.
    else
      g.drawString("Testing is off.", 10, 260); // Пункт Testing не отмечен
  }

  public static void main(String[] args) {
    MenuDemo appwin = new MenuDemo();

    appwin.setSize(new Dimension(250, 300));
    appwin.setTitle("MenuDemo");
    appwin.setVisible(true);
  }

  // Внутренний класс для обработки событий действий и элементов в меню.
  class MyMenuHandler implements ActionListener, ItemListener {

    // Обработать события действий.
    public void actionPerformed(ActionEvent ae) {
      msg = "You selected ";  // Выбран пункт меню ...
```

```
    String arg = ae.getActionCommand();
    if(arg.equals("New...")) msg += "New.";
    else if(arg.equals("Open...")) msg += "Open.";
    else if(arg.equals("Close")) msg += "Close.";
    else if(arg.equals("Edit")) msg += "Edit.";
    else if(arg.equals("Cut")) msg += "Cut.";
    else if(arg.equals("Copy")) msg += "Copy.";
    else if(arg.equals("Paste")) msg += "Paste.";
    else if(arg.equals("First")) msg += "First.";
    else if(arg.equals("Second")) msg += "Second.";
    else if(arg.equals("Third")) msg += "Third.";
    else if(arg.equals("Debug")) msg += "Debug.";
    else if(arg.equals("Testing")) msg += "Testing.";
    repaint();
  }
  // Обработать события элементов.
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }
 }
}
```

Пример вывода из программы MenuDemo показан на рис. 27.16.

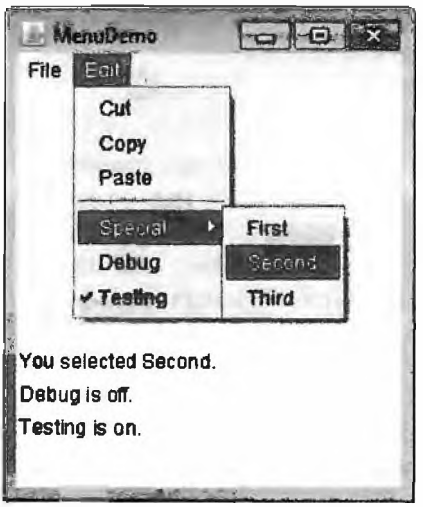


Рис. 27.16. Пример вывода из программы MenuDemo

Существует еще один класс, связанный с меню, который может показаться интересным: PopupMenu. Он работает аналогично классу Menu, но создает меню, которое может отображаться в определенном месте. Класс PopupMenu предлагает гибкую и полезную альтернативу для организации некоторых типов меню.

Диалоговые окна

Часто возникает необходимость в размещении набора связанных элементов управления внутри *диалогового окна*. Диалоговые окна применяются главным образом для ввода данных пользователем и нередко являются дочерними в отношении окна верхнего уровня. Диалоговые окна не имеют панелей меню, но в остальном они функционируют как фреймовые окна. (Скажем, элементы управления добавляются к ним аналогично добавлению элементов управления во фреймовые окна.) Диалоговые окна могут быть *модальными* или *немодальными*. Когда модальное диалоговое окно активно, получить доступ к другим окнам в программе (за исключением дочерних окон диалогового окна) не удастся, пока это диалоговое окно не будет закрыто. Когда немодальное диалоговое окно активно, фокус ввода может быть перемещен в другое окно программы, а потому другие части программы остаются активными и доступными. Начиная с версии JDK 6, модальные диалоговые окна можно создавать с тремя различными типами модальности, которые определены в перечислении `Dialog.ModalityType`. По умолчанию принимается вариант `APPLICATION_MODAL`, предотвращающий использование других окон верхнего уровня в приложении. Он представляет собой традиционный тип модальности. Другие типы модальности — `DOCUMENT_MODAL` и `TOOLKIT_MODAL`. Также предусмотрен тип `MODELESS`.

В библиотеке AWT диалоговые окна имеют тип `Dialog`. Вот два часто применяемых конструктора класса `Dialog`:

```
Dialog(Frame parentWindow, boolean mode)
Dialog(Frame parentWindow, String title, boolean mode)
```

В `parentWindow` указывается владелец диалогового окна. Если `mode` имеет значение `true`, тогда диалоговое окно использует стандартный тип модальности, а если `false`, то диалоговое окно будет немодальным. В `title` можно передать заголовок диалогового окна. В общем случае создается подкласс `Dialog`, к которому добавляется функциональность, необходимая приложению.

Далее представлена модифицированная версия предыдущей программы `MenuDemo`, которая теперь в случае выбора пункта меню **New** (Создать) отображает немодальное диалоговое окно. Обратите внимание, что при закрытии диалогового окна вызывается метод `dispose()`. Он определен в классе `Window` и освобождает все системные ресурсы, связанные с диалоговым окном.

```
// Демонстрация использования диалогового окна.

import java.awt.*;
import java.awt.event.*;

class DialogDemo extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;
  SampleDialog myDialog;
```

```
public DialogDemo() {
  // Создать диалоговое окно.
  myDialog = new SampleDialog(this, "New Dialog Box");

  // Создать панель меню и добавить ее во фрейм.
  MenuBar mbar = new MenuBar();
  setMenuBar(mbar);

  // Создать пункты меню.
  Menu file = new Menu("File");
  MenuItem item1, item2, item3, item4, item5;
  file.add(item1 = new MenuItem("New..."));
  file.add(item2 = new MenuItem("Open..."));
  file.add(item3 = new MenuItem("Close"));
  file.add(item4 = new MenuItem("-"));
  file.add(item5 = new MenuItem("Quit..."));
  mbar.add(file);

  Menu edit = new Menu("Edit");
  MenuItem item6, item7, item8, item9;
  edit.add(item6 = new MenuItem("Cut"));
  edit.add(item7 = new MenuItem("Copy"));
  edit.add(item8 = new MenuItem("Paste"));
  edit.add(item9 = new MenuItem("-"));

  Menu sub = new Menu("Special");
  MenuItem item10, item11, item12;
  sub.add(item10 = new MenuItem("First"));
  sub.add(item11 = new MenuItem("Second"));
  sub.add(item12 = new MenuItem("Third"));
  edit.add(sub);

  // Создать переключаемые пункты меню.
  debug = new CheckboxMenuItem("Debug");
  edit.add(debug);
  test = new CheckboxMenuItem("Testing");
  edit.add(test);

  mbar.add(edit);

  // Создать объект для обработки событий действий и элементов.
  MyMenuHandler handler = new MyMenuHandler();

  // Зарегистрировать для получения этих событий.
  item1.addActionListener(handler);
  item2.addActionListener(handler);
  item3.addActionListener(handler);
  item4.addActionListener(handler);
  item6.addActionListener(handler);
  item7.addActionListener(handler);
  item8.addActionListener(handler);
  item9.addActionListener(handler);
  item10.addActionListener(handler);
  item11.addActionListener(handler);
  item12.addActionListener(handler);
  debug.addItemListener(handler);
  test.addItemListener(handler);
```

```
    // Использовать лямбда-выражение для обработки выбора пункта Quit.
    item5.addActionListener((ae) -> System.exit(0));

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    g.drawString(msg, 10, 220);

    if(debug.getState())
      g.drawString("Debug is on.", 10, 240);
    else
      g.drawString("Debug is off.", 10, 240);

    if(test.getState())
      g.drawString("Testing is on.", 10, 260);
    else
      g.drawString("Testing is off.", 10, 260);
  }

  public static void main(String[] args) {
    DialogDemo appwin = new DialogDemo();

    appwin.setSize(new Dimension(250, 300));
    appwin.setTitle("DialogDemo");
    appwin.setVisible(true);
  }

  // Внутренний класс для обработки событий действий и элементов в меню.
  class MyMenuHandler implements ActionListener, ItemListener {

    // Обработать события действий.
    public void actionPerformed(ActionEvent ae) {
      msg = "You selected ";
      String arg = ae.getActionCommand();

      if(arg.equals("New...")) {
        msg += "New.";
        myDialog.setVisible(true);
      }
      else if(arg.equals("Open...")) msg += "Open.";
      else if(arg.equals("Close")) msg += "Close.";
      else if(arg.equals("Edit")) msg += "Edit.";
      else if(arg.equals("Cut")) msg += "Cut.";
      else if(arg.equals("Copy")) msg += "Copy.";
      else if(arg.equals("Paste")) msg += "Paste.";
      else if(arg.equals("First")) msg += "First.";
      else if(arg.equals("Second")) msg += "Second.";
      else if(arg.equals("Third")) msg += "Third.";
      else if(arg.equals("Debug")) msg += "Debug.";
      else if(arg.equals("Testing")) msg += "Testing.";

      repaint();
    }
```

```
    // Обработать события элементов.
    public void itemStateChanged(ItemEvent ie) {
      repaint();
    }
  }
}
// Создать подкласс Dialog.
class SampleDialog extends Dialog {
  SampleDialog(Frame parent, String title) {
    super(parent, title, false);
    setLayout(new FlowLayout());
    setSize(300, 200);
    add(new Label("Press this button:"));  // Нажмите на эту кнопку
    Button b;
    add(b = new Button("Cancel"));
    b.addActionListener((ae) -> dispose());
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        dispose();
      }
    });
  }
  public void paint(Graphics g) {
    g.drawString("This is in the dialog box", 20, 80);
                // Строка в диалоговом окне
  }
}
```

Пример вывода из программы `DialogDemo` показан на рис. 27.17.

Рис. 27.17. Пример вывода из программы `DialogDemo`

Совет. Попробуйте самостоятельно определить диалоговые окна для других пунктов меню.

Несколько слов о переопределении метода `paint()`

Прежде чем завершить исследование элементов управления AWT, имеет смысл затронуть тему переопределения метода `paint()`. Хотя это и не относится к простым примерам применения библиотеки AWT, показанным в книге, при переопределении метода `paint()` бывают случаи, когда необходимо вызывать реализацию `paint()` из суперкласса. Таким образом, в некоторых программах придется использовать следующую схему для `paint()`:

```
public void paint(Graphics g) {
  // Код перерисовки этого окна.
  // Вызов paint() из суперкласса.
  super.paint(g);
}
```

В Java существуют два основных типа компонентов: тяжеловесные и легковесные. Тяжеловесный компонент имеет собственное окно, которое называется *равноправным*. Легковесный компонент полностью реализован в коде Java и пользуется окном, предоставленным его предком. Все элементы управления AWT, которые были описаны и применялись в главе, являются тяжеловесными. Однако если контейнер содержит какие-то легковесные компоненты (т.е. имеет легковесные дочерние компоненты), то в переопределенной версии `paint()` для такого контейнера должен присутствовать вызов `super.paint()`. Наличие вызова `super.paint()` гарантирует, что любые легковесные дочерние компоненты, такие как легковесные элементы управления, будут правильно нарисованы. Если тип дочернего компонента не особо ясен, тогда для его установления можно вызвать метод `isLightweight()`, определенный в классе `Component`, который возвращает `true`, если компонент легковесный, или `false` в противном случае.

ГЛАВА

28 Изображения

В настоящей главе рассматривается класс Image и пакет java.awt.image. Вместе они обеспечивают поддержку *обработки изображений* (т.е. отображения и манипулирования графическими изображениями). *Изображение* представляет собой всего лишь прямоугольный графический объект. Изображения являются ключевым компонентом веб-дизайна. Фактически включение дескриптора <img> в браузер Mosaic разработчиками из NCSA (National Center for Supercomputer Applications — Национальный центр по использованию суперкомпьютеров) стало своего рода катализатором, который содействовал началу бурного роста веб-сети в 1993 году. Дескриптор <img> применялся для встраивания изображений в поток гипертекста. Язык Java расширяет эту базовую концепцию, позволяя управлять изображениями программным образом. По причине важности данного аспекта в Java предоставляется обширная поддержка для работы с изображениями.

Изображения поддерживаются классом Image, который входит в состав пакета java.awt. Манипулирование изображениями осуществляется с помощью классов из пакета java.awt.image. Количество классов и интерфейсов для обработки изображений, определенных в java.awt.image, настолько велико, что исследовать их все в одной главе невозможно. Взамен внимание будет сосредоточено только на тех классах и интерфейсах, которые формируют основу работы с изображениями. В главе обсуждаются следующие классы из пакета java.awt.image:

CropImageFilter	MemoryImageSource
FilteredImageSource	PixelGrabber
ImageFilter	RGBImageFilter

Кроме того, будут использоваться интерфейсы ImageConsumer и ImageProducer.

Форматы файлов

Изначально изображения для веб-сети могли иметь только формат GIF. Формат изображений GIF был создан CompuServe в 1987 году, чтобы сделать

возможным просмотр изображений в онлайновом режиме, и потому он хорошо подходил для Интернета. Каждое изображение GIF может содержать не более 256 цветов. Такое ограничение заставило крупнейших поставщиков браузеров в 1995 году добавить поддержку изображений JPEG. Формат JPEG был создан группой экспертов в области фотографии для хранения изображений с полным цветовым спектром и непрерывными тонами. При правильном создании изображение в формате JPEG может иметь гораздо более высокую точность и быть лучше сжатым по сравнению с тем же изображением в формате GIF. Еще одним форматом файла стал PNG, который тоже является альтернативой GIF. Практически во всех разрабатываемых программах не придется заботиться или обращать внимание на то, какой именно формат применяется. Классы изображений Java абстрагируют различия посредством четко определенного интерфейса.

Основы работы с изображениями: создание, загрузка и отображение

При работе с изображениями выполняются три общие операции: создание изображения, загрузка изображения и отображение изображения. Для ссылки на изображения в памяти и на изображения, которые должны быть загружены из внешних источников, в Java используется класс `Image`. Таким образом, в Java предоставляются способы создания нового объекта изображения и способы его загрузки, а также средства, с помощью которых изображение можно отображать. Давайте рассмотрим их по очереди.

Создание объекта изображения

Может показаться, что изображение в памяти создается примерно так:

```
Image test = new Image(200, 100);  // Ошибка -- не работает!
```

На самом деле ситуация иная. В конечном итоге изображения должны рисоваться в окне, но класс `Image` не обладает достаточной информацией о своей среде, чтобы создать надлежащий формат данных для экрана, поэтому в классе `Component` из пакета `java.awt` определен фабричный метод `createImage()`, который применяется для создания объектов `Image`. (Вспомните, что все компоненты AWT являются подклассами `Component` и потому поддерживают упомянутый метод.)

Метод `createImage()` имеет две формы:

```
Image createImage(ImageProducer imgProd)
Image createImage(int width, int height)
```

Первая форма метода возвращает изображение, созданное с помощью поставщика изображений `imgProd`, где передается объект класса, реализующего интерфейс `ImageProducer`. (Поставщики изображений обсуждаются позже в главе.) Вторая форма возвращает пустое изображение с шириной `width` и высотой `height`. Вот пример:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
```

В примере создается экземпляр Canvas, после чего вызывается метод createImage() для фактического создания объекта Image. Пока что изображение пустое. Позже будет показано, как записывать в него данные.

Загрузка изображения

Другой способ получить изображение связан с его загрузкой либо из файла в локальной файловой системе, либо из URL. Здесь будет использоваться локальная файловая система. Самый простой способ загрузки изображения предусматривает применение одного из статических методов класса ImageIO, который обеспечивает обширную поддержку чтения и записи изображений. Он находится в пакете javax.imageio. Начиная с версии JDK 9, пакет javax.imageio входит в состав модуля java.desktop. Метод, загружающий изображение, называется read(). Используемая здесь форма приведена ниже:

```
static BufferedImage read(File imageFile) throws IOException
```

В imageFile указывается файл, содержащий изображение. Метод возвращает ссылку на изображение в виде объекта класса BufferedImage, который является подклассом Image, включающим буфер. Если файл не содержит допустимое изображение, тогда возвращается null.

Отображение изображения

Полученное изображение можно отобразить с помощью метода drawImage(), который определен в классе Graphics. Он имеет несколько форм, и одна из них будет применяться далее в главе:

```
boolean drawImage(Image imgObj, int left, int top, ImageObserver imgOb)
```

Показанный выше метод drawImage() отображает изображение, переданное в imgObj, в местоположении с левым верхним углом, заданным в left и top. Параметр imgOb представляет собой ссылку на класс, реализующий интерфейс наблюдателя изображения по имени ImageObserver, который реализуется всеми компонентами AWT (и Swing). *Наблюдатель изображения* — это объект, который способен отслеживать изображение во время его загрузки. Когда наблюдатель изображения не нужен, imgOb может иметь значение null.

В действительности с использованием методов read() и drawImage() загружать и отображать изображение довольно легко. Ниже показана программа, в которой загружается и отображается одиночное изображение. Вместо файла Lilies.jpg можно указать любое предпочитаемое изображение (при условии, что оно находится в том же каталоге, что и программа). Пример вывода представлен на рис. 28.1.

```
// Загрузка и отображение изображения.

import java.awt.*;
import java.awt.event.*;
import javax.imageio.*;
```

```
import java.io.*;

public class SimpleImageLoad extends Frame {
  Image img;

  public SimpleImageLoad() {

    try {
      File imageFile = new File("Lilies.jpg");

      // Загрузить изображение.
      img = ImageIO.read(imageFile);
    } catch (IOException exc) {
      System.out.println("Не удалось загрузить файл изображения.");
      System.exit(0);
    }

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    g.drawImage(img, getInsets().left, getInsets().top, null);
  }

  public static void main(String[] args) {
    SimpleImageLoad appwin = new SimpleImageLoad();

    appwin.setSize(new Dimension(400, 365));
    appwin.setTitle("SimpleImageLoad");
    appwin.setVisible(true);
  }
}
```

Рис. 28.1. Пример вывода из программы SimpleImageLoad

Двойная буферизация

Изображения удобны не только для хранения фотографий, как только что было продемонстрировано — их также допускается применять в качестве внеэкранных поверхностей рисования, позволяющих отображать любое изображение, включая текст и графику, во внеэкранный буфер, который можно отобразить позднее. Преимущество такого подхода заключается в том, что изображение будет видно только после завершения его формирования. Вычерчивание сложного изображения может занимать несколько миллисекунд и более, что пользователь часто воспринимает как мигание или мерцание. В итоге визуализация кажется пользователю более медленным процессом, чем есть на самом деле. Использование внеэкранного изображения для уменьшения мерцания называется *двойной буферизацией*, поскольку экран считается первым буфером для пикселей, а внеэкранное изображение — вторым буфером, где можно подготовить пиксели для отображения.

Ранее в главе вы видели, как создавать пустой объект `Image`. Теперь вы узнаете, каким образом рисовать на таком объекте `Image` вместо экрана. Вспомните, что для применения любого метода визуализации Java необходим объект `Graphics`. Удобно то, что объект `Graphics`, который можно использовать для рисования на изображении, доступен через метод `getGraphics()`. Вот фрагмент кода, демонстрирующий создание нового изображения, получение его графического контекста и заполнение всего изображения пикселями красного цвета:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
Graphics gc = test.getGraphics();
gc.setColor(Color.red);
gc.fillRect(0, 0, 200, 100);
```

После конструирования и заполнения внеэкранного изображения его по-прежнему не будет видно. Чтобы отобразить изображение, понадобится вызвать метод `drawImage()`. Ниже приведен пример рисования изображения, сопряженного с высокими затратами времени, который иллюстрирует эффект от двойной буферизации, касающийся восприятия времени рисования:

```
// Демонстрация использования внеэкранного буфера.

import java.awt.*;
import java.awt.event.*;

public class DoubleBuffer extends Frame {
  int gap = 3;
  int mx, my;
  boolean flicker = true;
  Image buffer = null;
  int w = 400, h = 400;

  public DoubleBuffer() {
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent me) {
```

```
        mx = me.getX();
        my = me.getY();
        flicker = false;
        repaint();
      }
      public void mouseMoved(MouseEvent me) {
        mx = me.getX();
        my = me.getY();
        flicker = true;
        repaint();
      }
    });
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
  public void paint(Graphics g) {
    Graphics screengc = null;
    if(!flicker) {
      screengc = g;
      g = buffer.getGraphics();
    }
    g.setColor(Color.blue);
    g.fillRect(0, 0, w, h);
    g.setColor(Color.red);
    for (int i=0; i<w; i+=gap)
      g.drawLine(i, 0, w-i, h);
    for (int i=0; i<h; i+=gap)
      g.drawLine(0, i, w, h-i);
    g.setColor(Color.black);
    g.drawString("Press mouse button to double buffer", 10, h/2);
                // Нажмите кнопку мыши, чтобы удвоить буфер
    g.setColor(Color.yellow);
    g.fillOval(mx - gap, my - gap, gap*2+1, gap*2+1);
    if(!flicker) {
      screengc.drawImage(buffer, 0, 0, null);
    }
  }
  public void update(Graphics g) {
    paint(g);
  }
  public static void main(String[] args) {
    DoubleBuffer appwin = new DoubleBuffer();
    appwin.setSize(new Dimension(400, 400));
    appwin.setTitle("DoubleBuffer");
    appwin.setVisible(true);
    // Создать внеэкранный буфер.
    appwin.buffer = appwin.createImage(appwin.w, appwin.h);
  }
}
```

Несмотря на относительную простоту, программа DoubleBuffer имеет сложный метод paint(), который окрашивает фон в синий цвет, выводит на нем красный муаровый узор, поверх рисует текст черного цвета и, наконец, вычерчивает желтый круг с координатами центра mx, my. Кроме того, для отслеживания положения указателя мыши переопределяются методы mouseMoved() и mouseDragged(), которые идентичны за исключением установки булевской переменной flicker. В методе mouseMoved() переменная flicker устанавливается в true, а в методе mouseDragged() — в false. В итоге метод repaint() вызывается со значением flicker, равным true, когда указатель мыши перемещается (но никакая кнопка мыши не нажата), и false, когда указатель мыши перетаскивается при любой нажатой кнопке.

Если метод paint() вызывается, когда переменная flicker установлена в true, то на экране можно наблюдать выполнение каждой операции рисования. Но если кнопка мыши нажата и метод paint() вызывается, когда переменная flicker установлена в false, тогда картина будет совершенно другой. Метод paint() заменяет ссылку g типа Graphics графическим контекстом, который ссылается на внеэкранный холст, т.е. buffer, созданный в main(), и затем все операции рисования становятся невидимыми. В конце метода paint() просто вызывается drawImage(), чтобы отобразить результаты выполнения всех методов рисования.

Вывод программы DoubleBuffer представлен на рис. 28.2. Слева показано, как выглядит экран, когда кнопка мыши не нажата. Легко заметить, что изображение находится в процессе перерисовки. На экране справа видно, что в случае нажатия кнопки мыши изображение получается полным благодаря двойной буферизации.

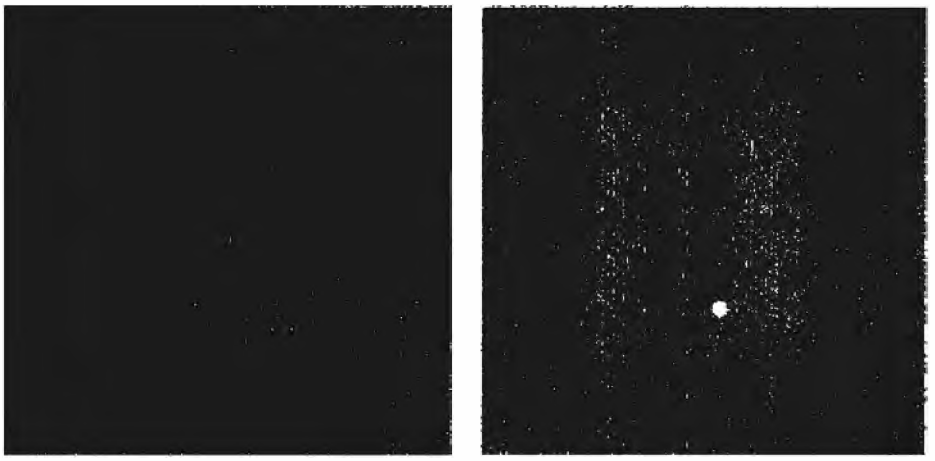


Рис. 28.2. Вывод программы DoubleBuffer с двойной буферизацией (слева) и без нее (справа)

ImageProducer

Интерфейс ImageProducer позволяет объектам производить данные для изображений. Объект, реализующий интерфейс ImageProducer, будет предоставлять целочисленные или байтовые массивы с данными изображений, и создавать объекты Image. Как было показано ранее, одна из форм метода createImage() принимает в качестве аргумента объект ImageProducer.

В пакете java.awt.image содержатся два поставщика изображений: MemoryImageSource и FilteredImageSource. Далее в главе исследуется класс MemoryImageSource на примере создания нового объекта Image из сгенерированных данных.

MemoryImageSource

Класс MemoryImageSource создает новое изображение из массива данных. В нем определено несколько конструкторов, и вот тот, который будет применяться:

```
MemoryImageSource(int width, int height, int[] pixel, int offset,
                  int scanLineWidth)
```

Объект MemoryImageSource создается из массива целых чисел, указанного в pixel, со стандартной цветовой моделью RGB, чтобы произвести данные для объекта Image. В стандартной цветовой модели пиксель определен как целое число с альфа-каналом, красной, зеленой и синей составляющими (0xAARRGGBB). Значение альфа-канала представляет степень прозрачности пикселя. Полной прозрачности соответствует 0, а полной непрозрачности — 255. Ширина и высота результирующего изображения передаются в width и height. Стартовая точка в массиве pixel, с которой должно начинаться чтение данных, задается в offset. Ширина строки развертки (часто совпадающая с шириной изображения) указывается в scanLineWidth.

В следующем примере создается объект MemoryImageSource с использованием вариации простого алгоритма (побитового исключающего ИЛИ координат X и Y каждого пикселя) из книги Джерарда Дж. Хольцмана *Beyond Photography: The Digital Darkroom* (Prentice Hall, 1988).

```
// Создание изображения в памяти.

import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;

public class MemoryImageGenerator extends Frame {
  Image img;
  int w = 512;
  int h = 512;

  public MemoryImageGenerator() {
    int[] pixels = new int[w * h];
    int i = 0;
```

```
    for(int y=0; y<h; y++) {
      for(int x=0; x<w; x++) {
        int r = (x^y)&0xff;
        int g = (x*2^y*2)&0xff;
        int b = (x*4^y*4)&0xff;
        pixels[i++] = (255 << 24) | (r << 16) | (g << 8) | b;
      }
    }
    img = createImage(new MemoryImageSource(w, h, pixels, 0, w));
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
  public void paint(Graphics g) {
    g.drawImage(img, getInsets().left, getInsets().top, null);
  }
  public static void main(String[] args) {
    MemoryImageGenerator appwin = new MemoryImageGenerator();
    appwin.setSize(new Dimension(400, 400));
    appwin.setTitle("MemoryImageGenerator");
    appwin.setVisible(true);
  }
}
```

Данные для нового объекта `MemoryImageSource` создаются в конструкторе. Для хранения значений пикселей предусмотрен массив целых чисел, а сами данные генерируются во вложенных циклах `for`, где значения `r`, `g` и `b` сдвигаются, представляя пиксель в массиве `pixels`. Наконец, вызывается метод `createImage()`, которому в качестве параметра передается новый экземпляр `MemoryImageSource`, созданный из низкоуровневых данных пикселей. Результирующее изображение показано на рис. 28.3.

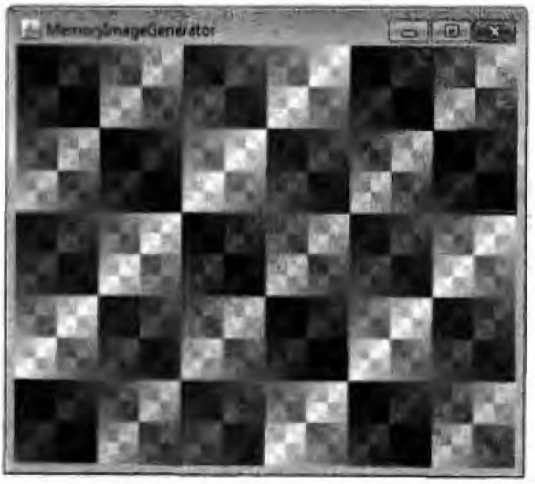


Рис. 28.3. Пример вывода из программы `MemoryImageGenerator`

ImageConsumer

Интерфейс ImageConsumer предназначен для объектов, которым необходимо извлекать данные пикселей из изображений и предоставлять их как данные другого вида. Очевидно, он является противоположностью описанного ранее интерфейса ImageProducer. Объект, реализующий интерфейс ImageConsumer, будет создавать целочисленные или байтовые массивы, представляющие пиксели из объекта Image. Ниже рассматривается класс PixelGrabber, который обеспечивает простую реализацию интерфейса ImageConsumer.

PixelGrabber

Класс PixelGrabber определен в пакете java.lang.image и является противоположностью класса MemoryImageSource. Вместо того чтобы формировать изображение из массива значений пикселей, он получает существующее изображение и *захватывает* из него массив пикселей. Прежде чем можно будет применять PixelGrabber, нужно создать целочисленный массив, достаточно большой для хранения данных пикселей, и затем создать экземпляр класса PixelGrabber, передав его конструктору прямоугольник, который требуется захватить. В заключение необходимо вызвать метод grabPixels() на экземпляре PixelGrabber.

Вот конструктор класса PixelGrabber, который используется в этой главе:

```
PixelGrabber(Image imgObj, int left, int top, int width, int height,
             int[] pixel, int offset, int scanLineWidth)
```

В imgObj указывается объект, пиксели которого захватываются. Значения left и top задают левый верхний угол, а width и height — размеры прямоугольника, из которого будут получаться пиксели. Пиксели сохраняются в массиве pixel, начиная с offset. В scanLineWidth передается ширина строки развертки (которая часто совпадает с шириной изображения).

Ниже показаны сигнатуры метода grabPixels():

```
boolean grabPixels()
  throws InterruptedException
boolean grabPixels(long milliseconds)
  throws InterruptedException
```

Обе формы метода возвращают true в случае успеха или false в случае неудачи. Во второй форме в milliseconds указывается промежуток времени в миллисекундах, в течение которого метод будет ожидать пиксели. Обе формы генерируют исключение InterruptedException, если выполнение прерывается другим потоком.

Далее приведен пример захвата пикселей из изображения и создания гистограммы яркости пикселей. Гистограмма просто отражает количество пикселей, которые имеют определенную яркость для всех настроек яркости от 0 до 255. После рисования изображения поверх него вычерчивается гистограмма.

```
// Демонстрация использования PixelGraber.
import java.awt.* ;
import java.awt.event.*;
import java.awt.image.*;
import javax.imageio.*;
import java.io.*;

public class HistoGrab extends Frame {
  Dimension d;
  Image img;
  int iw, ih;
  int[] pixels;
  int[] hist = new int[256];
  int max_hist = 0;
  Insets ins;

  public HistoGrab() {

    try {
      File imageFile = new File("Lilies.jpg");

      // Загрузить изображение.
      img = ImageIO.read(imageFile);

      iw = img.getWidth(null);
      ih = img.getHeight(null);
      pixels = new int[iw * ih];
      PixelGrabber pg = new PixelGrabber(img, 0, 0, iw, ih,
                                         pixels, 0, iw);
      pg.grabPixels();
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
      return;
    } catch (IOException exc) {
      System.out.println("Не удалось загрузить файл изображения.");
      System.exit(0);
    }

    for (int i=0; i<iw*ih; i++) {
      int p = pixels[i];
      int r = 0xff & (p >> 16);
      int g = 0xff & (p >> 8);
      int b = 0xff & (p);
      int y = (int) (.33 * r + .56 * g + .11 * b);
      hist[y]++;
    }
    for (int i=0; i<256; i++) {
      if (hist[i] > max_hist)
        max_hist = hist[i];
    }

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
```

```
  public void paint(Graphics g) {
    // Получить вставки кромки/заголовка.
    ins = getInsets();
    g.drawImage(img, ins.left, ins.top, null);
    int x = (iw - 256) / 2;
    int lasty = ih - ih * hist[0] / max_hist;
    for (int i=0; i<256; i++, x++) {
      int y = ih - ih * hist[i] / max_hist;
      g.setColor(new Color(i, i, i));
      g.fillRect(x+ins.left, y+ins.top, 1, ih-y);
      g.setColor(Color.red);
      g.drawLine((x-1)+ins.left,lasty+ins.top,x+ins.left,y+ins.top);
      lasty = y;
    }
  }
  public static void main(String[] args) {
    HistoGrab appwin = new HistoGrab();
    appwin.setSize(new Dimension(400, 380));
    appwin.setTitle("HistoGrab");
    appwin.setVisible(true);
  }
}
```

Пример изображения и связанной с ним гистограммы можно видеть на рис. 28.4.

Рис. 28.4. Пример вывода из программы `HistoGrab`

ImageFilter

При наличии пары интерфейсов ImageProducer и ImageConsumer, а также реализующих эти интерфейсы конкретных классов MemoryImageSource и PixelGrabber можно создать произвольный набор трансляционных фильтров, которые получают пиксели из источника, изменяют их и передают произвольному потребителю. Такой механизм аналогичен способу создания конкретных классов из абстрактных классов ввода-вывода InputStream, OutputStream, Reader и Writer (описанных в главе 22). Эта потоковая модель для изображений дополняется введением класса ImageFilter.

В число подклассов ImageFilter внутри пакета java.awt.image входят AreaAveragingScaleFilter, CropImageFilter, ReplicateScaleFilter и RGBImageFilter. Существует также реализация интерфейса ImageProducer по имени FilteredImageSource, которая помещает ImageFilter в оболочку ImageProducer с целью фильтрации создаваемых им пикселей. Экземпляр FilteredImageSource можно применять как объект реализации ImageProducer в вызовах метода createImage() во многом аналогично возможности использования BufferedInputStream в качестве InputStream.

В главе исследуются два фильтра: CropImageFilter и RGBImageFilter.

CropImageFilter

Класс CropImageFilter выполняет фильтрацию источника изображения для извлечения прямоугольной области. Одна из ситуаций, когда полезен фильтр CropImageFilter, связана с желанием получить несколько небольших изображений из одного большего исходного изображения. Загрузка двадцати изображений размером 2 Кбайт занимает гораздо больше времени, чем загрузка одного изображения размером 40 Кбайт, куда включено множество кадров анимации. Если все части изображения имеют одинаковые размеры, то их можно легко извлечь с помощью CropImageFilter, разобрав блок после запуска программы. Ниже показан пример, в котором создаются 16 изображений, полученных из одного изображения. Затем элементы мозаики перемешиваются путем замены случайной пары из 16 изображений 32 раза.

```
// Демонстрация использования CropImageFilter.

import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;
import javax.imageio.*;
import java.io.*;

public class TileImage extends Frame {
  Image img;
  Image[] cell = new Image[4*4];
  int iw, ih;
  int tw, th;
```

```
  public TileImage() {
    try {
      File imageFile = new File("Lilies.jpg");

      // Загрузить изображение.
      img = ImageIO.read(imageFile);

      iw = img.getWidth(null);
      ih = img.getHeight(null);
      tw = iw / 4;
      th = ih / 4;

      CropImageFilter f;
      FilteredImageSource fis;

      for (int y=0; y<4; y++) {
        for (int x=0; x<4; x++) {
          f = new CropImageFilter(tw*x, th*y, tw, th);
          fis = new FilteredImageSource(img.getSource(), f);
          int i = y*4+x;
          cell[i] = createImage(fis);
        }
      }
      for (int i=0; i<32; i++) {
        int si = (int)(Math.random() * 16);
        int di = (int)(Math.random() * 16);
        Image tmp = cell[si];
        cell[si] = cell[di];
        cell[di] = tmp;
      }
    } catch (IOException exc) {
      System.out.println("Не удалось загрузить файл изображения.");
      System.exit(0);
    }
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
  public void paint(Graphics g) {
    for (int y=0; y<4; y++) {
      for (int x=0; x<4; x++) {
        g.drawImage(cell[y*4+x], x * tw + getInsets().left,
                    y * th + getInsets().top, null);
      }
    }
  }
  public static void main(String[] args) {
    TileImage appwin = new TileImage();

    appwin.setSize(new Dimension(420, 420));
    appwin.setTitle("TileImage");
    appwin.setVisible(true);
  }
}
```

На рис. 28.5 показано изображение цветков, перемешанное посредством программы TileImage.

Рис. 28.5. Пример вывода из программы TileImage

RGBImageFilter

Класс RGBImageFilter применяется для преобразования одного изображения в другое, пиксель за пикселем, с попутной трансформацией цветов. Фильтр RGBImageFilter можно использовать для осветления изображения, увеличения его контрастности или даже для преобразования его в оттенки серого.

Чтобы продемонстрировать применение класса RGBImageFilter, пришлось разработать довольно сложный пример, в котором используется стратегия динамически подключаемых фильтров обработки изображений. Для обобщенной фильтрации изображений был создан интерфейс, позволяющий программе просто загружать фильтры изображений во время выполнения, не зная заранее обо всех фильтрах подобного рода. В состав примера входит главный класс ImageFilterDemo, интерфейс PlugInFilter и служебный класс LoadedImage. Кроме того, предусмотрены три фильтра, Grayscale, Invert и Contrast, которые манипулируют цветовым пространством исходного изображения с применением объектов RGBImageFilter. Наконец, есть еще два класса, Blur и Sharpen, выполняющие более сложные фильтры свертки, которые изменяют данные пикселей на основе пикселей, окружающих каждый пиксель в исходных данных. Классы Blur и Sharpen являются подклассами абстрактного вспомогательного класса Convolver. Ниже по очереди рассматриваются все части примера.

ImageFilterDemo.java

ImageFilterDemo — главный класс для примеров фильтров изображений. В нем используется стандартный экземпляр BorderLayout с объектом Panel в позиции South для размещения кнопок, которые будут представлять каждый фильтр. Объект Label занимает позицию North и предназначен для отображения информационных сообщений о ходе фильтрации. В позиции Center находится изображение (которое инкапсулировано в подклассе LoadedImage класса Canvas, описанном ниже).

Метод actionPerformed() интересен тем, что применяет метку кнопки в качестве имени класса фильтра, который он загружает. Данный метод надежен и выполняет подходящее действие, если кнопка не соответствует надлежащему классу, реализующему PlugInFilter.

```
// Демонстрация использования фильтров изображений.
import java.awt.*;
import java.awt.event.*;
import javax.imageio.*;
import java.io.*;
import java.lang.reflect.*;
public class ImageFilterDemo extends Frame implements ActionListener {
  Image img;
  PlugInFilter pif;
  Image fimg;
  Image curImg;
  LoadedImage lim;
  Label lab;
  Button reset;

  // Имена фильтров.
  String[] filters = {"Grayscale", "Invert", "Contrast", "Blur", "Sharpen"};
  public ImageFilterDemo() {
    Panel p = new Panel();
    add(p, BorderLayout.SOUTH);

    // Создать кнопку Reset (Сброс).
    reset = new Button("Reset");
    reset.addActionListener(this);
    p.add(reset);

    // Добавить кнопки фильтров.
    for(String fstr: filters) {
      Button b = new Button(fstr);
      b.addActionListener(this);
      p.add(b);
    }

    // Создать верхнюю кнопку.
    lab = new Label("");
    add(lab, BorderLayout.NORTH);

    // Загрузить изображение.
    try {
      File imageFile = new File("Lilies.jpg");
      img = ImageIO.read(imageFile);
```

```
    } catch (IOException exc) {
      System.out.println("Не удалось загрузить файл изображения.");
      System.exit(0);
    }
    // Получить объект LoadedImage и расположить его по центру.
    lim = new LoadedImage(img);
    add(lim, BorderLayout.CENTER);
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
  public void actionPerformed(ActionEvent ae) {
    String a = "";
    try {
      a = ae.getActionCommand();
      if (a.equals("Reset")) {
        lim.set(img);
        lab.setText("Normal");              // Нормальное
      }
      else {
        // Получить выбранный фильтр.
        pif = (PlugInFilter)
              (Class.forName(a)).getConstructor().newInstance();
        fimg = pif.filter(this, img);
        lim.set(fimg);
        lab.setText("Filtered: " + a);      // Отфильтрованное
      }
      repaint();
    } catch (ClassNotFoundException e) {
      lab.setText(a + " not found");        // не найдено
      lim.set(img);
      repaint();
    } catch (InstantiationException e) {
      lab.setText("couldn't new " + a);     // не удается создать
    } catch (IllegalAccessException e) {
      lab.setText("no access: " + a);       // нет доступа
    } catch (NoSuchMethodException | InvocationTargetException e) {
      lab.setText("Filter creation error: " + e); // Ошибка создания фильтра
    }
  }
  public static void main(String[] args) {
    ImageFilterDemo appwin = new ImageFilterDemo();
    appwin.setSize(new Dimension(420, 420));
    appwin.setTitle("ImageFilterDemo");
    appwin.setVisible(true);
  }
}
```

На рис. 28.6 показан внешний вид программы при ее первой загрузке.

Рис. 28.6. Пример нормального вывода из программы `ImageFilterDemo`

PlugInFilter.java

`PlugInFilter` — простой интерфейс, используемый для абстрактной фильтрации изображений. У него есть только один метод `filter()`, который принимает фрейм и исходное изображение и возвращает новое изображение, которое каким-то образом было отфильтровано.

```
interface PlugInFilter {
  java.awt.Image filter(java.awt.Frame f, java.awt.Image in);
}
```

LoadedImage.java

Класс `LoadedImage` — удобный подкласс `Canvas`, который ведет себя надлежащим образом под управлением диспетчера компоновки, потому что переопределяет методы `getPreferredSize()` и `getMinimumSize()`. Кроме того, у него есть метод `set()`, который можно применять для установки нового объекта `Image`, подлежащего отображению на холсте. Именно так отображается отфильтрованное изображение после завершения работы подключаемого фильтра.

```
import java.awt.*;

public class LoadedImage extends Canvas {
  Image img;

  public LoadedImage(Image i) {
    set(i);
  }
```

```
  void set(Image i) {
    img = i;
    repaint();
  }
  public void paint(Graphics g) {
    if (img == null) {
      g.drawString("no image", 10, 30);  // изображение отсутствует
    } else {
      g.drawImage(img, 0, 0, this);
    }
  }
  public Dimension getPreferredSize() {
    return new Dimension(img.getWidth(this), img.getHeight(this));
  }
  public Dimension getMinimumSize() {
    return getPreferredSize();
  }
}
```

Grayscale.java

Фильтр `Grayscale` является подклассом `RGBImageFilter`, т.е. объект `Grayscale` можно передавать в качестве параметра типа `ImageFilter` конструктору `FilteredImageSource`. В нем потребуется лишь переопределить метод `filterRGB()`, чтобы изменить входные значения цвета. Метод `filterRGB()` принимает значения красной, зеленой и синей составляющих и вычисляет яркость пикселя с использованием коэффициента преобразования цвета в яркость, принятого NTSC (National Television Standards Committee — Национальный комитет по телевизионным стандартам). Затем он просто возвращает серый пиксель с такой же яркостью, как у исходного цвета.

```
// Фильтр яркости.
import java.awt.*; import java.awt.image.*;
class Grayscale extends RGBImageFilter implements PlugInFilter {
  public Grayscale() {}
  public Image filter(Frame f, Image in) {
    return f.createImage(new FilteredImageSource(in.getSource(), this));
  }
  public int filterRGB(int x, int y, int rgb) {
    int r = (rgb >> 16) & 0xff;
    int g = (rgb >> 8) & 0xff;
    int b = rgb & 0xff;
    int k = (int) (.56 * g + .33 * r + .11 * b);
    return (0xff000000 | k << 16 | k << 8 | k);
  }
}
```

Invert.java

Фильтр Invert тоже довольно прост. Он разделяет значение цвета на красную, зеленую и синюю составляющие, после чего инвертирует их, вычитая из 255. Инвертированные величины упаковываются обратно в значение пикселя и возвращаются.

```
// Фильтр инвертирования цветов.
import java.awt.*;
import java.awt.image.*;

class Invert extends RGBImageFilter implements PlugInFilter {
  public Invert() {}

  public Image filter(Frame f, Image in) {
    return f.createImage(new FilteredImageSource(in.getSource(), this));
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = 0xff - (rgb >> 16) & 0xff;
    int g = 0xff - (rgb >> 8) & 0xff;
    int b = 0xff - rgb & 0xff;
    return (0xff000000 | r << 16 | g << 8 | b);
  }
}
```

На рис. 28.7 показано изображение, полученное в результате обработки фильтром Invert.

Рис. 28.7. Применение фильтра Invert в программе ImageFilterDemo

Contrast.java

Фильтр Contrast очень похож на Grayscale за исключением того, что переопределенная в нем версия метода filterRGB() немного сложнее. Алгоритм, который он использует для повышения контрастности, принимает значения красной, зеленой и синей составляющих по отдельности и умножает их на 1.2, если их яркость выше 128. Если же яркость ниже 128, тогда они делятся на 1.2. Увеличенные значения должным образом фиксируются на уровне 255 с помощью метода multclamp().

```
// Фильтр контрастности.
import java.awt.*;
import java.awt.image.*;
public class Contrast extends RGBImageFilter implements PlugInFilter {
  public Image filter(Frame f, Image in) {
    return f.createImage(new FilteredImageSource(in.getSource(), this));
  }
  private int multclamp(int in, double factor) {
    in = (int) (in * factor);
    return in > 255 ? 255 : in;
  }
  double gain = 1.2;
  private int cont(int in) {
    return (in < 128) ? (int)(in/gain) : multclamp(in, gain);
  }
  public int filterRGB(int x, int y, int rgb) {
    int r = cont((rgb >> 16) & 0xff);
    int g = cont((rgb >> 8) & 0xff);
    int b = cont(rgb & 0xff);
    return (0xff000000 | r << 16 | g << 8 | b);
  }
}
```

На рис. 28.8 представлено изображение после обработки фильтром Contrast.

Рис. 28.8. Использование фильтра Contrast в программе ImageFilterDemo

Convolver.java

Абстрактный класс Convolver поддерживает базовую обработку фильтра свертки, реализуя интерфейс ImageConsumer для перемещения исходных пикселей в массив по имени imgpixels. Он также создает второй массив с именем newimgpixels для отфильтрованных данных. Фильтры свертки производят выборку небольшого прямоугольника пикселей вокруг каждого пикселя изображения, называемого *ядром свертки*. Такая область размером 3×3 применяется в демонстрационной программе для принятия решения о том, как изменить центральный пиксель в этой области.

На заметку! Причина, по которой фильтр не может модифицировать массив imgpixels на месте, связана с тем, что следующий пиксель в строке развертки может попытаться использовать первоначальное значение предыдущего пикселя, который мог оказаться только что отфильтрованным.

Два конкретных подкласса, описанные в последующих разделах, просто реализуют метод convolve() с применением массива imgpixels для исходных данных и массива newimgpixels для сохранения результата.

```
// Фильтр свертки.
import java.awt.*;
import java.awt.image.*;

abstract class Convolver implements ImageConsumer, PlugInFilter {
  int width, height;
  int[] imgpixels, newimgpixels;
  boolean imageReady = false;

  abstract void convolve();  // Здесь находится фильтр...

  public Image filter(Frame f, Image in) {
    imageReady = false;

    in.getSource().startProduction(this);
    waitForImage();
    newimgpixels = new int[width*height];

    try {
      convolve();
    } catch (Exception e) {
      System.out.println("Отказ Convolver: " + e);
      e.printStackTrace();
    }
    return f.createImage(
      new MemoryImageSource(width, height, newimgpixels, 0, width));
  }

  synchronized void waitForImage() {
    try {
      while(!imageReady)
        wait();
    } catch (Exception e) {
      System.out.println("Interrupted");  // Прервано
    }
  }
```

```
    public void setProperties(java.util.Hashtable<?,?> dummy) {}
    public void setColorModel(ColorModel dummy) {}
    public void setHints(int dummy) {}

    public synchronized void imageComplete(int dummy) {
      imageReady = true;
      notifyAll();
    }

    public void setDimensions(int x, int y) {
      width = x;
      height = y;
      imgpixels = new int[x*y];
    }

    public void setPixels(int x1, int y1, int w, int h,
              ColorModel model, byte[] pixels, int off, int scansize) {
      int pix, x, y, x2, y2, sx, sy;

      x2 = x1+w;
      y2 = y1+h;
      sy = off;
      for(y=y1; y<y2; y++) {
        sx = sy;
        for(x=x1; x<x2; x++) {
          pix = model.getRGB(pixels[sx++]);
          if((pix & 0xff000000) == 0)
            pix = 0x00ffffff;
          imgpixels[y*width+x] = pix;
        }
        sy += scansize;
      }
    }

    public void setPixels(int x1, int y1, int w, int h,
              ColorModel model, int[] pixels, int off, int scansize) {
      int pix, x, y, x2, y2, sx, sy;

      x2 = x1+w;
      y2 = y1+h;
      sy = off;
      for(y=y1; y<y2; y++) {
        sx = sy;
        for(x=x1; x<x2; x++) {
          pix = model.getRGB(pixels[sx++]);
          if((pix & 0xff000000) == 0)
            pix = 0x00ffffff;
          imgpixels[y*width+x] = pix;
        }
        sy += scansize;
      }
    }
  }
```

На заметку! В пакете `java.awt.image` предлагается встроенный фильтр свертки по имени `ConvolveOp`. Возможно, имеет смысл изучить его возможности самостоятельно.

Blur.java

Фильтр Blur является подклассом Convolver. Он просматривает каждый пиксель в массиве с исходным изображением imgpixels и вычисляет среднее значение окружающего пиксель прямоугольника 3×3, которое становится выходным пикселем в массиве newimgpixels.

```
public class Blur extends Convolver {
  public void convolve() {
    for(int y=1; y<height-1; y++) {
      for(int x=1; x<width-1; x++) {
        int rs = 0;
        int gs = 0;
        int bs = 0;
        for(int k=-1; k<=1; k++) {
          for(int j=-1; j<=1; j++) {
            int rgb = imgpixels[(y+k)*width+x+j];
            int r = (rgb >> 16) & 0xff;
            int g = (rgb >> 8) & 0xff;
            int b = rgb & 0xff;
            rs += r;
            gs += g;
            bs += b;
          }
        }
        rs /= 9;
        gs /= 9;
        bs /= 9;
        newimgpixels[y*width+x] = (0xff000000 | rs << 16 | gs << 8 | bs);
      }
    }
  }
}
```

На рис. 28.9 показано изображение после обработки фильтром Blur.

Рис. 28.9. Использование фильтра Blur в программе ImageFilterDemo

Sharpen.java

Фильтр Sharpen также является подклассом Convolver и (в какой-то степени) противоположностью Blur. Он проходит через каждый пиксель в массиве с исходным изображением imgpixels и вычисляет среднее значение окружающего его прямоугольника 3×3, не считая центральный пиксель. К соответствующему выходному пикселю в newimgpixels добавляется разница между центральным пикселем и окружающим средним значением. По существу, если пиксель на 30 единиц ярче своего окружения, тогда он делается ярче еще на 30 единиц. Если же пиксель на 10 единиц темнее, то он делается темнее еще на 10 единиц. В итоге резкие края выделяются, а гладкие области остаются без изменений.

```
public class Sharpen extends Convolver {
  private final int clamp(int c) {
    return (c > 255 ? 255 : (c < 0 ? 0 : c));
  }

  public void convolve() {
    int r0=0, g0=0, b0=0;

    for(int y=1; y<height-1; y++) {
      for(int x=1; x<width-1; x++) {
        int rs = 0;
        int gs = 0;
        int bs = 0;

        for(int k=-1; k<=1; k++) {
          for(int j=-1; j<=1; j++) {
            int rgb = imgpixels[(y+k)*width+x+j];
            int r = (rgb >> 16) & 0xff;
            int g = (rgb >> 8) & 0xff;
            int b = rgb & 0xff;
            if(j == 0 && k == 0) {
              r0 = r;
              g0 = g;
              b0 = b;
            } else {
              rs += r;
              gs += g;
              bs += b;
            }
          }
        }

        rs >>= 3;
        gs >>= 3;
        bs >>= 3;
        newimgpixels[y*width+x] = (0xff000000 | clamp(r0+r0-rs) << 16 |
                                   clamp(g0+g0-gs) << 8 | clamp(b0+b0-bs));
      }
    }
  }
}
```

На рис. 28.10 представлено изображение после обработки фильтром Sharpen.

Рис. 28.10. Применение фильтра `Sharpen` в программе `ImageFilterDemo`

Дополнительные классы для обработки изображений

Помимо классов для обработки изображений, описанных в настоящей главе, пакет `java.awt.image` предлагает ряд других классов, которые обеспечивают расширенный контроль над процессом обработки и поддерживают более совершенные приемы формирования изображений. Кроме того, доступен пакет обработки изображений под названием `javax.imageio`. Он поддерживает чтение и запись изображений и имеет подключаемые модули для обработки разнообразных форматов изображений. Если вас интересует сложный графический вывод, тогда придется изучить дополнительные классы, находящиеся в пакетах `java.awt.image` и `javax.imageio`.

ГЛАВА

29 Утилиты параллелизма

Язык Java с самого начала предлагал встроенную поддержку многопоточности и синхронизации. Например, новые потоки можно создавать за счет реализации `Runnable` или расширения `Thread`, синхронизация доступна при использовании ключевого слова `synchronized`, а взаимодействие между потоками обеспечивается методами `wait()` и `notify()`, определенными в `Object`. В целом встроенная поддержка многопоточности была одним из самых важных нововведений языка Java и до сих пор остается одной из его сильных сторон.

Тем не менее, какой бы концептуально чистой ни была первоначальная поддержка многопоточности в Java, она не идеальна для всех приложений — в особенности для тех, которые интенсивно эксплуатируют несколько потоков. Скажем, исходная поддержка многопоточности не предоставляет такие высокоуровневые средства, как семафоры, пулы потоков и диспетчеры выполнения, которые облегчают создание в высокой степени параллельных программ.

Вначале важно пояснить, что многие программы на Java применяют многопоточность и, следовательно, являются "параллельными". Однако понятие *параллельной программы* в том виде, в каком оно используется в настоящей главе, относится к программе, в которой *широко и всеобъемлюще* задействованы одновременно выполняющиеся потоки. Примером такой программы может служить программа, которая применяет отдельные потоки для одновременного вычисления частичных результатов более крупного вычисления. В качестве еще одного примера можно назвать программу, координирующую действия нескольких потоков, каждый из которых пытается получить доступ к информации в базе данных. В таком случае доступ с целью только чтения может обрабатываться иначе, чем доступ, требующий возможностей чтения и записи.

Потребности параллельных программ начали удовлетворяться добавлением в JDK 5 утилит параллелизма, также обычно называемых *параллельным API*. Исходный набор утилит параллелизма предоставил множество функций, которые давно были нужны программистам, разрабатывающим параллельные приложения. Например, он предлагал синхронизаторы (вроде семафоров),

пулы потоков, диспетчеры выполнения, блокировки, несколько параллельных коллекций и упрощенный способ использования потоков для получения результатов вычислений.

Хотя первоначальный параллельный API был впечатляющим сам по себе, в версии JDK 7 он значительно расширился. Самым важным дополнением стала инфраструктура Fork/Join Framework, облегчающая создание программ, которые задействуют несколько процессоров (например, в многоядерных системах). Таким образом, она упрощает разработку программ, где две или более частей выполняются по-настоящему одновременно (т.е. параллельно), а не просто путем квантования времени. Несложно предположить, что параллельное выполнение способно значительно увеличить скорость выполнения определенных операций. Поскольку в наше время многоядерные системы — вполне обычное явление, включение Fork/Join Framework стало столь же своевременным, сколь и действенным. В JDK 8 инфраструктура Fork/Join Framework была усовершенствована.

Кроме того, в JDK 8 и JDK 9 появились функциональные средства, имеющие отношение к другим частям параллельного API. Таким образом, с годами параллельный API развивался и расширялся, чтобы соответствовать потребностям современной вычислительной среды.

Первоначальный параллельный API был довольно крупным, а дополнения, произведенные за прошедшие годы, существенно увеличили его размер. Как и следовало ожидать, многие проблемы, связанные с утилитами параллелизма, довольно сложны, и обсуждение всех аспектов выходит за рамки этой книги. Несмотря на вышесказанное, всем программистам важно иметь базовые практические знания ключевых аспектов параллельного API. Даже в программах, в которых параллелизм интенсивно не применяется, такие средства, как синхронизаторы, вызываемые потоки и исполнители, применимы в самых разнообразных ситуациях. Возможно, важнее всего то, что из-за роста количества многоядерных компьютеров решения, использующие Fork/Join Framework, становятся все более распространенными. По указанным причинам в главе представлен обзор ряда основных средств, определяемых утилитами параллелизма, и продемонстрировано несколько примеров их применения. Глава завершается введением в инфраструктуру Fork/Join Framework.

Пакеты параллельного API

Утилиты параллелизма содержатся в пакете `java.util.concurrent` и в двух его подпакетах: `java.util.concurrent.atomic` и `java.util.concurrent.locks`. Начиная с версии JDK 9, все они находятся в модуле `java.base`. Далее приведен краткий обзор содержимого упомянутых пакетов.

java.util.concurrent

В пакете java.util.concurrent определены ключевые функциональные средства, которые являются альтернативами встроенным подходам к синхронизации и взаимодействию между потоками, в том числе:

- синхронизаторы;
- исполнители;
- параллельные коллекции;
- инфраструктура Fork/Join Framework.

Синхронизаторы предлагают высокоуровневые способы синхронизации взаимодействия между несколькими потоками. Классы синхронизаторов, определенные в java.util.concurrent, кратко описаны в табл. 29.1.

Таблица 29.1. Классы синхронизаторов из пакета java.util.concurrent

Класс	Описание
Semaphore	Реализует классический семафор
CountDownLatch	Ожидает до тех пор, пока не произойдет указанное количество событий
CyclicBarrier	Позволяет группе потоков ожидать в заранее определенной точке выполнения
Exchanger	Осуществляет обмен данными между двумя потоками
Phaser	Синхронизирует потоки, проходящие через несколько стадий операции

Обратите внимание, что каждый синхронизатор обеспечивает решение задачи синхронизации конкретного вида, что позволяет его оптимизировать для предполагаемого использования. Сразу после появления Java такие типы объектов синхронизации приходилось создавать вручную. Параллельный API стандартизировал их и сделал доступными для всех программистов на Java.

Исполнители управляют выполнением потока. На вершине иерархии исполнителей находится интерфейс Executor, который применяется для запуска потока. Интерфейс ExecutorService расширяет Executor и предоставляет методы, управляющие выполнением. Существуют три реализации интерфейса ExecutorService: ThreadPoolExecutor, ScheduledThreadPoolExecutor и ForkJoinPool. В пакете java.util.concurrent также определен служебный класс Executors, который включает несколько статических методов, упрощающих создание различных исполнителей.

С исполнителями связаны интерфейсы Future и Callable. Интерфейс Future содержит значение, которое возвращается потоком после его выполнения. Таким образом, значение определяется "в будущем" (отсюда и имя Future), когда поток завершится. Интерфейс Callable определяет поток, который возвращает значение.

В пакете java.util.concurrent определено несколько классов параллельных коллекций, в том числе ConcurrentHashMap, ConcurrentLinkedQueue и CopyOnWriteArrayList. Они предлагают параллельные альтернативы связанным с ними классам из инфраструктуры Collections Framework.

Инфраструктура Fork/Join Framework поддерживает параллельное программирование. Ее основными классами являются ForkJoinTask, ForkJoinPool, RecursiveTask и RecursiveAction.

Для более совершенной обработки синхронизации потоков в java.util.concurrent определено перечисление TimeUnit.

Начиная с версии JDK 9, пакет java.util.concurrent также включает подсистему, предлагающую средства, с помощью которых можно контролировать поток данных. Она основана на классе Flow и вложенных интерфейсах Flow.Subscriber, Flow.Publisher, Flow.Processor и Flow.Subscription.

Хотя подробное обсуждение подсистемы на базе Flow выходит за рамки главы, здесь приведено ее краткое описание. Интерфейс Flow и его вложенные интерфейсы поддерживают спецификацию *реактивных потоков*, определяющую средства, с помощью которых потребитель данных может преодолеть запрет поставщика на их обработку. При таком подходе данные производятся *издателем* и потребляются *подписчиком*. Контроль достигается за счет реализации формы *обратного давления*.

java.util.concurrent.atomic

Пакет java.util.concurrent.atomic облегчает использование переменных в параллельной среде. Он предоставляет средства для эффективного обновления значений переменных без применения блокировок. Задача решается за счет использования классов вроде AtomicInteger и AtomicLong, а также методов наподобие compareAndSet(), decrementAndGet() и getAndSet(). Упомянутые методы выполняются как одна непрерываемая операция.

java.util.concurrent.locks

Пакет java.util.concurrent.locks предоставляет альтернативные средства взамен применения синхронизированных методов. В основе средств лежит интерфейс Lock, который определяет базовый механизм, используемый для получения и отказа в доступе к объекту. В число ключевых методов входят lock(), tryLock() и unlock(). Преимущество указанных методов — более высокая степень контроля над синхронизацией.

В оставшемся материале главы будут подробно рассматриваться составляющие части параллельного API.

Использование объектов синхронизации

Объекты синхронизации поддерживаются классами Semaphore, CountDownLatch, CyclicBarrier, Exchanger и Phaser. Все вместе они позволяют легко справиться с несколькими ранее сложными ситуациями, свя-

занными с синхронизацией. Они также применимы к широкому спектру программ — даже к тем, в которых присутствует лишь ограниченный параллелизм. Так как объекты синхронизации представляют интерес почти для всех программ Java, каждый из них анализируется по отдельности.

Semaphore

Наиболее узнаваемым объектом синхронизации является `Semaphore`, реализующий классический семафор. Семафор управляет доступом к общему ресурсу с помощью счетчика. Если счетчик больше нуля, то доступ разрешен. Если он равен нулю, тогда доступ запрещен. Счетчик подсчитывает *разрешения*, которые открывают доступ к общему ресурсу. Таким образом, для доступа к ресурсу поток должен получить разрешение от семафора.

Обычно для использования семафора поток, желающий иметь доступ к общему ресурсу, пытается получить разрешение. Если счетчик семафора больше нуля, тогда поток получает разрешение, что приводит к декрементированию счетчика семафора. В противном случае поток будет заблокирован до тех пор, пока не будет получено разрешение. Когда потоку больше не нужен доступ к общему ресурсу, он освобождает разрешение, что приводит к инкрементированию счетчика семафора. При наличии другого потока, ожидающего разрешение, в данный момент этот другой поток получит разрешение. Описанный механизм реализован классом `Semaphore`.

В классе `Semaphore` определены два конструктора:

```
Semaphore(int num)
Semaphore(int num, boolean how)
```

В `num` указывается начальное количество разрешений. Следовательно, `num` задает число потоков, которые одновременно могут получить доступ к общему ресурсу. Если значение `num` равно 1, тогда в любой момент времени доступ к ресурсу может получить только один поток. По умолчанию ожидающие потоки получают разрешение в неопределенном порядке. Установка параметра `how` в `true` обеспечивает получение разрешений ожидающими потоками в порядке запрашивания ими доступа. Чтобы запросить доступ, потребуется вызвать метод `acquire()`, имеющий две формы:

```
void acquire() throws InterruptedException
void acquire(int num) throws InterruptedException
```

Первая форма метода запрашивает одно разрешение, а вторая форма — `num` разрешений. Чаще всего применяется первая форма. Если разрешение не может быть выдано в момент вызова, тогда вызывающий поток приостанавливается до тех пор, пока разрешение не станет доступным.

Чтобы освободить разрешение, понадобится вызвать метод `release()`, имеющий две формы:

```
void release()
void release(int num)
```

Первая форма освобождает одно разрешение, а вторая — `num` разрешений.

Чтобы использовать семафор для управления доступом к ресурсу, каждый поток, который заинтересован в ресурсе, перед доступом к нему должен вызвать метод `acquire()`. После завершения работы с ресурсом поток обязан вызвать метод `release()`. Применение семафора иллюстрируется в следующем примере:

```
// Простой пример использования семафора.
import java.util.concurrent.*;
class SemDemo {
  public static void main(String[] args) {
    Semaphore sem = new Semaphore(1);
    new Thread(new IncThread(sem, "A")).start();
    new Thread(new DecThread(sem, "B")).start();
  }
}
// Общий ресурс.
class Shared {
  static int count = 0;
}
// Поток выполнения, который инкрементирует счетчик (переменную count).
class IncThread implements Runnable {
  String name;
  Semaphore sem;
  IncThread(Semaphore s, String n) {
    sem = s;
    name = n;
  }
  public void run() {
    System.out.println("Запуск " + name);
    try {
      // Для начала получить разрешение.
      System.out.println(name + " ожидает разрешения.");
      sem.acquire();
      System.out.println(name + " получил разрешение.");
      // Теперь получить доступ к общему ресурсу.
      for(int i=0; i < 5; i++) {
        Shared.count++;
        System.out.println(name + ": " + Shared.count);
        // Разрешить переключение контекста по возможности.
        Thread.sleep(10);
      }
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
    // Освободить разрешение.
    System.out.println(name + " освободил разрешение.");
    sem.release();
  }
}
```

```
// Поток выполнения, который декрементирует счетчик (переменную count).
class DecThread implements Runnable {
  String name;
  Semaphore sem;

  DecThread(Semaphore s, String n) {
    sem = s;
    name = n;
  }

  public void run() {
    System.out.println("Запуск " + name);
    try {
      // Для начала получить разрешение.
      System.out.println(name + " ожидает разрешения.");
      sem.acquire();
      System.out.println(name + " получил разрешение.");

      // Теперь получить доступ к общему ресурсу.
      for(int i=0; i < 5; i++) {
        Shared.count--;
        System.out.println(name + ": " + Shared.count);

        // Разрешить переключение контекста по возможности.
        Thread.sleep(10);
      }
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    // Освободить разрешение.
    System.out.println(name + " освободил разрешение.");
    sem.release();
  }
}
```

Ниже показан вывод из программы (порядок выполнения потоков может варьироваться):

```
Запуск A
A ожидает разрешения.
A получил разрешение.
A: 1
Запуск B
B ожидает разрешения.
A: 2
A: 3
A: 4
A: 5
A освободил разрешение.
B получил разрешение.
B: 4
B: 3
B: 2
B: 1
B: 0
B освободил разрешение.
```

Семафор используется в программе для управления доступом к переменной count, которая является статической переменной в классе Shared. Значение Shared.count пять раз инкрементируется в методе run() класса IncThread и пять раз декрементируется в методе run() класса DecThread. Чтобы предотвратить одновременный доступ этих двух потоков к переменной Shared.count, доступ открывается только после получения разрешения от управляющего семафора. По окончании доступа разрешение освобождается. Таким образом, доступ к Shared.count будет иметь только один поток за раз, что легко заметить в выводе из программы.

Обратите внимание в классах IncThread и DecThread на вызов метода sleep() внутри run(). Он предназначен для того, чтобы "подтвердить" синхронизацию доступа к Shared.count с помощью семафора. В методе run() вызов sleep() заставляет вызывающий поток приостанавливаться между доступами к Shared.count, что обычно дает возможность выполняться второму потоку. Тем не менее, из-за семафора второй поток должен ожидать, пока первый поток не освободит разрешение, что происходит только после завершения всех обращений первого потока. Таким образом, значение Shared.count пять раз инкрементируется в потоке IncThread и пять раз декрементируется в потоке DecThread. Инкрементирование и декрементирование *не* перемешиваются.

Без семафора доступ к переменной Shared.count обоими потоками происходил бы одновременно, а инкрементирование и декрементирование оказались бы смешанными. Чтобы удостовериться в этом, попробуйте закомментировать вызовы acquire() и release(). Запустив программу, вы заметите, что доступ к переменной Shared.count больше не синхронизируется, и каждый поток обращается к ней, как только получает квант времени.

Хотя многие варианты применения семафора в той же степени просты, как было продемонстрировано в предыдущей программе, возможны и более интересные ситуации. В показанном далее примере модифицируется программа с производителем и потребителем из главы 11. Теперь в ней с помощью двух семафоров упорядочивается выполнение потоков производителя и потребителя, так что за каждым вызовом put() следует соответствующий вызов get().

```
// Реализация производителя и потребителя, в которой
// для управления синхронизацией используются семафоры.

import java.util.concurrent.Semaphore;

class Q {
  int n;

  // Начать с недоступным семафором потребителя.
  static Semaphore semCon = new Semaphore(0);
  static Semaphore semProd = new Semaphore(1);
```

```
  void get() {
    try {
      semCon.acquire();
    } catch(InterruptedException e) {
      System.out.println("Перехвачено исключение InterruptedException");
    }
    System.out.println("Получено: " + n);
    semProd.release();
  }
  void put(int n) {
    try {
      semProd.acquire();
    } catch(InterruptedException e) {
      System.out.println("Перехвачено исключение InterruptedException");
    }
    this.n = n;
    System.out.println("Отправлено: " + n);
    semCon.release();
  }
}
class Producer implements Runnable {
  Q q;
  Producer(Q q) {
    this.q = q;
  }
  public void run() {
    for(int i=0; i < 20; i++)
      q.put(i);
  }
}
class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
    this.q = q;
  }
  public void run() {
    for(int i=0; i < 20; i++)
      q.get();
  }
}
class ProdCon {
  public static void main(String[] args) {
    Q q = new Q();
    new Thread(new Consumer(q), "Consumer").start();
    new Thread(new Producer(q), "Producer").start();
  }
}
```

Вот фрагмент вывода из программы:

```
Отправлено: 0
Получено: 0
Отправлено: 1
Получено: 1
Отправлено: 2
Получено: 2
Отправлено: 3
Получено: 3
Отправлено: 4
Получено: 4
Отправлено: 5
Получено: 5
.
.
.
```

Несложно заметить, что вызовы методов `put()` и `get()` синхронизированы, т.е. за каждым вызовом `put()` следует вызов `get()`, и никакие значения не пропускаются. Без семафоров несколько вызовов `put()` не совпали бы с вызовами `get()`, приводя к пропуску значений. (Чтобы удостовериться в этом, удалите код семафора и взгляните на результаты.)

Последовательность вызовов `put()` и `get()` обрабатывается двумя семафорами: `semProd` и `semCon`. Прежде чем метод `put()` сможет выпустить значение, он должен получить разрешение от `semProd`. После установки значения метод `put()` освобождает `semCon`. Прежде чем метод `get()` сможет потребить значение, он должен получить разрешение от `semCon`. После потребления значения метод `get()` освобождает `semProd`. Такой механизм "взаимных уступок" гарантирует, что за каждым вызовом `put()` должен следовать вызов `get()`.

Обратите внимание, что `semCon` инициализируется без доступных разрешений, что гарантирует выполнение метода `put()` первым. Возможность установки начального состояния синхронизации является одним из наиболее мощных аспектов семафора.

CountDownLatch

Иногда может требоваться, чтобы поток ожидал, пока не произойдет одно или несколько событий. Для решения такой задачи параллельный API предлагает класс `CountDownLatch`. Экземпляр `CountDownLatch` изначально создается со счетчиком событий, которые должны произойти, прежде чем защелка будет освобождена. При каждом возникновении события счетчик декрементируется. Когда счетчик достигает нуля, защелка открывается.

В классе `CountDownLatch` определен следующий конструктор:

```
CountDownLatch(int num)
```

В `num` указывается количество событий, которые должны произойти, чтобы защелка открылась.

Для организации ожидания защелки в потоке вызывается метод await(), формы которого показаны ниже:

```
void await() throws InterruptedException
boolean await(long wait, TimeUnit tu) throws InterruptedException
```

Первая форма метода await() организует ожидание до тех пор, пока счетчик, связанный с вызывающим объектом CountDownLatch, не достигнет нуля. Вторая форма обеспечивает ожидание только в течение периода времени, указанного в wait. Единицы периода wait задаются в параметре tu, который является объектом перечисления TimeUnit. (Тип TimeUnit описан далее в главе.) Метод await() возвращает false, если лимит времени ожидания исчерпан, и true, если обратный отсчет достиг нуля.

Чтобы сигнализировать о событии, потребуется вызвать метод countDown():

```
void countDown()
```

Каждый вызов countDown() декрементирует счетчик, ассоциированный с вызывающим объектом. Использование CountDownLatch демонстрируется в приведенной ниже программе, где создается защелка, для открытия которой требуется пять событий.

```
// Пример использования CountDownLatch.
import java.util.concurrent.CountDownLatch;
class CDLDemo {
  public static void main(String[] args) {
    CountDownLatch cdl = new CountDownLatch(5);
    System.out.println("Начало");
    new Thread(new MyThread(cdl)).start();
    try {
      cdl.await();
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
    System.out.println("Конец");
  }
}
class MyThread implements Runnable {
  CountDownLatch latch;
  MyThread(CountDownLatch c) {
    latch = c;
  }
  public void run() {
    for(int i = 0; i<5; i++) {
      System.out.println(i);
      latch.countDown(); // декрементирует счетчик
    }
  }
}
```

Вот вывод, генерируемый программой:

```
Начало
0
1
2
3
4
Конец
```

В методе `main()` создается объект `CountDownLatch` по имени `cdl` с начальным значением счетчика, равным пяти. Затем создается экземпляр `MyThread`, который начинает выполнение нового потока. Обратите внимание, что в качестве параметра конструктору `MyThread` передается `cdl` и сохраняется в переменной экземпляра `latch`. Затем в главном потоке вызывается `await()` на `cdl`, что приводит к приостановке выполнения главного потока до тех пор, пока счетчик `cdl` не будет декрементирован пять раз.

Внутри метода `run()` класса `MyThread` создается цикл, который повторяется пять раз. На каждой итерации производится вызов метода `countDown()` на объекте `latch`, который ссылается на `cdl` из `main()`. После пятой итерации защелка открывается, что позволяет возобновить выполнение главного потока.

Таким образом, `CountDownLatch` — мощный, но простой в использовании объект синхронизации, подходящий в тех случаях, когда поток должен ожидать возникновения одного или нескольких событий.

CyclicBarrier

Во время параллельного программирования нередко возникает ситуация, когда набор из двух или более потоков должен ожидать в заданной точке выполнения, пока все потоки в наборе не достигнут этой точки. Чтобы справиться с такой ситуацией, параллельный API предлагает класс `CyclicBarrier`. Он позволяет определить объект синхронизации, который приостанавливается до тех пор, пока указанное количество потоков не достигнет точки барьера.

Класс `CyclicBarrier` имеет два конструктора:

```
CyclicBarrier(int numThreads)
CyclicBarrier(int numThreads, Runnable action)
```

В `numThreads` передается количество потоков, которые должны достичь барьера, прежде чем выполнение продолжится. Во второй форме конструктора в `action` указывается поток, который будет выполняться при достижении барьера.

Существует общая процедура применения `CyclicBarrier`. Первым делом понадобится создать объект `CyclicBarrier`, указав количество потоков, для которых будет организовано ожидание. Когда каждый поток достигает барьера, он должен вызвать метод `await()` на объекте `CyclicBarrier`, что приведет к приостановке выполнения потока до тех пор, пока все остальные потоки тоже не вызовут `await()`. Как только указанное количество потоков

достигнет барьера, метод await() возвратит управление и выполнение возобновится. Кроме того, если в action указано действие, то выполнится этот поток.

Есть две формы метода await():

```
int await() throws InterruptedException, BrokenBarrierException
int await(long wait, TimeUnit tu)
  throws InterruptedException, BrokenBarrierException, TimeoutException
```

Первая форма организует ожидание до тех пор, пока все потоки не доберутся до точки барьера. Во второй форме ожидание длится только в течение периода времени, указанного посредством wait в единицах, заданных в tu. Обе формы метода await() возвращают значение, отражающее порядок, в котором потоки достигают точки барьера. Первый поток возвращает значение, равное количеству ожидающих потоков минус один. Последний поток возвращает ноль.

Ниже приведен пример, иллюстрирующий использование CyclicBarrier. В нем организуется ожидание до тех пор, пока набор из трех потоков не достигнет барьера, после чего выполняется поток, указанный в BarAction.

```
// Пример использования CyclicBarrier.

import java.util.concurrent.*;

class BarDemo {
  public static void main(String[] args) {
    CyclicBarrier cb = new CyclicBarrier(3, new BarAction());

    System.out.println("Начало");

    new Thread(new MyThread(cb, "A")).start();
    new Thread(new MyThread(cb, "B")).start();
    new Thread(new MyThread(cb, "C")).start();
  }
}

// Поток выполнения, в котором применяется CyclicBarrier.
class MyThread implements Runnable {
  CyclicBarrier cbar;
  String name;

  MyThread(CyclicBarrier c, String n) {
    cbar = c;
    name = n;
  }

  public void run() {
    System.out.println(name);
    try {
      cbar.await();
    } catch (BrokenBarrierException exc) {
      System.out.println(exc);
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
  }
}
```

```
// Объект этого класса вызывается, когда заканчивается CyclicBarrier.
class BarAction implements Runnable {
  public void run() {
    System.out.println("Барьер достигнут!");
  }
}
```

Вот вывод, генерируемый программой (порядок выполнения потоков может варьироваться):

```
Начало
A
B
C
Барьер достигнут!
```

Объект `CyclicBarrier` можно применять многократно, т.к. он освобождает ожидающие потоки каждый раз, когда указанное количество потоков вызывает метод `await()`. Например, если метод `main()` из предыдущей программы привести к следующему виду:

```
public static void main(String[] args) {
  CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );

  System.out.println("Начало");

  new Thread(new MyThread(cb, "A")).start();
  new Thread(new MyThread(cb, "B")).start();
  new Thread(new MyThread(cb, "C")).start();
  new Thread(new MyThread(cb, "X")).start();
  new Thread(new MyThread(cb, "Y")).start();
  new Thread(new MyThread(cb, "Z")).start();
}
```

то будет получен показанный далее вывод (порядок выполнения потоков может варьироваться):

```
Начало
A
B
C
Барьер достигнут!
X
Y
Z
Барьер достигнут!
```

Как видно в предыдущем примере, класс `CyclicBarrier` предлагает упрощенное решение ранее сложной проблемы.

Exchanger

Пожалуй, самым интересным классом синхронизации является `Exchanger`. Он предназначен для упрощения обмена данными между двумя потоками. Класс `Exchanger` работает на удивление просто: он всего лишь ожидает, пока

два отдельных потока не вызовут его метод exchange(), в случае чего он осуществляет обмен данными, которые предоставляют потоки. Такой механизм элегантен и несложен в использовании. Случаи применения Exchanger представить легко. Скажем, один поток может готовить буфер для информации из сети, а другой — заполнять его информацией, полученной через сетевое подключение. Два потока работают совместно, так что каждый раз, когда требуется новый буфер, выполняется обмен.

Класс Exchanger является обобщенным классом со следующим определением:

```
Exchanger<V>
```

В V указывается тип обмениваемых данных.

В классе Exchanger определен единственный метод exchange(), который имеет две формы:

```
V exchange(V objRef) throws InterruptedException
V exchange(V objRef, long wait, TimeUnit tu)
  throws InterruptedException, TimeoutException
```

В objRef передается ссылка на данные, подлежащие обмену. Метод возвращает данные, полученные из другого потока. Вторая форма exchange() позволяет указать период ожидания. Ключевым моментом метода exchange() является то, что он не достигнет успеха до тех пор, пока не будет вызван для одного и того же объекта Exchanger двумя отдельными потоками. Именно так метод exchange() синхронизирует обмен данными.

Далее приведен пример, в котором демонстрируется использование класса Exchanger. Сначала конструируются два потока. Один поток создает пустой буфер, который будет получать данные, помещаемые в него вторым потоком. Здесь данные представляют собой строку. Таким образом, первый поток меняет пустую строку на заполненную.

```
// Пример использования Exchanger.

import java.util.concurrent.Exchanger;

class ExgrDemo {
  public static void main(String[] args) {
    Exchanger<String> exgr = new Exchanger<String>();

    new Thread(new UseString(exgr)).start();
    new Thread(new MakeString(exgr)).start();
  }
}

// Поток, который конструирует строку.
class MakeString implements Runnable {
  Exchanger<String> ex;
  String str;

  MakeString(Exchanger<String> c) {
    ex = c;
    str = new String();
  }
```

```
    public void run() {
      char ch = 'A';

      for(int i = 0; i < 3; i++) {

        // Заполнить буфер.
        for(int j = 0; j < 5; j++)
          str += ch++;

        try {
          // Обменять полный буфер на пустой.
          str = ex.exchange(str);
        } catch(InterruptedException exc) {
          System.out.println(exc);
        }
      }
    }
  }

  // Поток, который использует строку.
  class UseString implements Runnable {
    Exchanger<String> ex;
    String str;
    UseString(Exchanger<String> c) {
      ex = c;
    }

    public void run() {

      for(int i=0; i < 3; i++) {
        try {
          // Обменять пустой буфер на полный.
          str = ex.exchange(new String());
          System.out.println("Получено: " + str);
        } catch(InterruptedException exc) {
          System.out.println(exc);
        }
      }
    }
  }
```

Вот вывод, генерируемый программой:

```
Получено: ABCDE
Получено: FGHIJ
Получено: KLMNO
```

В методе `main()` создается объект `Exchanger` для строк, который затем применяется для синхронизации обмена строками между классами `MakeString` и `UseString`. Класс `MakeString` заполняет строку данными, а класс `UseString` заменяет пустую строку полной, после чего отображает содержимое вновь созданной строки. Обмен пустыми и полными буферами синхронизируется с помощью метода `exchange()`, который вызывается в методе `run()` обоих классов.

Phaser

Еще один класс синхронизации называется Phaser. Его основная цель связана с обеспечением синхронизации потоков, которые представляют одну или несколько стадий активности. Например, может существовать набор потоков, реализующих три стадии приложения обработки заказов. На первой стадии отдельные потоки используются для верификации сведений о клиенте, проверки запасов и подтверждения цен. После завершения первой стадии вторая стадия имеет два потока, которые рассчитывают стоимость доставки и все подходящие налоги. Затем на заключительной стадии подтверждается оплата и определяется расчетное время доставки. Раньше для синхронизации множества потоков, задействованных в таком сценарии, требовалось приложить немало усилий. С появлением класса Phaser данный процесс стал намного проще.

Для начала полезно знать, что класс Phaser работает примерно так же, как описанный ранее класс CyclicBarrier, за исключением того, что он поддерживает несколько стадий. В итоге Phaser позволяет определить объект синхронизации, который ожидает завершения определенной стадии. Затем он переходит к следующей стадии, снова ожидая ее завершения. Важно понимать, что Phaser также можно применять для синхронизации только одной стадии. В этом отношении он действует во многом подобно CyclicBarrier. Однако его основное использование предусматривает синхронизацию нескольких стадий. В классе Phaser определены четыре конструктора, два из которых будут применяться далее в главе:

```
Phaser()
Phaser(int numParties)
```

Первая форма конструктора создает объект Phaser с количеством регистраций, равным нулю, а вторая форма устанавливает количество регистраций в numParties. Термин *сторона* часто используется в отношении объектов, которые регистрируются с помощью Phaser. Хотя обычно существует однозначное соответствие между количеством регистрируемых объектов и количеством синхронизируемых потоков, это не обязательно. В обоих случаях текущая стадия равна нулю, т.е. при создании объект Phaser изначально находится на нулевой стадии.

Давайте выясним, как работать с классом Phaser. Сначала нужно создать новый экземпляр Phaser. Затем понадобится зарегистрировать одну или несколько сторон посредством Phaser, либо вызвав метод register(), либо указав количество сторон в вызове конструктора. Для каждой зарегистрированной стороны необходимо обеспечить, чтобы объект Phaser ожидал до тех пор, пока все зарегистрированные стороны не завершат стадию. Сторона сигнализирует о прибытии вызовом одного из многочисленных методов, предлагаемых Phaser, таких как arrive() или arriveAndAwaitAdvance(). После того, как все стороны прибыли, стадия завершена, и объект Phaser может переходить к следующей стадии (при ее наличии) или закончить работу. Процесс будет подробно описан в последующих разделах.

Для регистрации сторон после конструирования объекта Phaser должен вызываться метод register():

```
int register()
```

Он возвращает номер стадии, для которой была зарегистрирована сторона. Сторона сигнализирует о том, что она завершила стадию, вызовом метода arrive() либо его вариации. Когда количество прибывших равно количеству зарегистрированных сторон, стадия завершается и объект Phaser переходит к следующей стадии (если она есть). Метод arrive() имеет следующую общую форму:

```
int arrive()
```

Данный метод сигнализирует о том, что сторона (обычно поток выполнения) выполнила некоторую задачу (или часть задачи). Он возвращает номер текущей стадии. Если объект Phaser закончил работу, тогда arrive() возвращает отрицательное значение. Метод arrive() не приостанавливает выполнение вызывающего потока, т.е. он не ждет завершения стадии. Метод arrive() должен вызываться только зарегистрированной стороной.

Метод arriveAndAwaitAdvance() применяется, когда нужно указать на завершение стадии и затем подождать, пока все остальные зарегистрированные стороны тоже ее завершат:

```
int arriveAndAwaitAdvance()
```

Он организует ожидание до тех пор, пока не прибудут все стороны, и возвращает номер следующей стадии или отрицательное значение, если объект Phaser закончил работу. Метод arriveAndAwaitAdvance() должен вызываться только зарегистрированной стороной.

Поток может прибыть, а затем отменить регистрацию, вызвав метод arriveAndDeregister():

```
int arriveAndDeregister()
```

Метод arriveAndDeregister() возвращает номер текущей стадии или отрицательное значение, если объект Phaser закончил работу. Он не ждет завершения стадии. Данный метод должен вызываться только зарегистрированной стороной.

Метод getPhase() позволяет получить номер текущей стадии:

```
final int getPhase()
```

При создании объекта Phaser первой стадией будет 0, второй — 1, третьей — 2 и т.д. Отрицательное значение возвращается, если вызывающий объект Phaser закончил работу.

Ниже показан пример, демонстрирующий класс Phaser в действии. В нем создаются три потока, с каждым из которых связаны три стадии. Для синхронизации каждой стадии используется объект Phaser.

```
// Пример использования Phaser.
import java.util.concurrent.*;
```

```
class PhaserDemo {
  public static void main(String[] args) {
    Phaser phsr = new Phaser(1);
    int curPhase;

    System.out.println("Начало");

    new Thread(new MyThread(phsr, "A")).start();
    new Thread(new MyThread(phsr, "B")).start();
    new Thread(new MyThread(phsr, "C")).start();

    // Ожидать завершения всеми потоками первой стадии.
    curPhase = phsr.getPhase();
    phsr.arriveAndAwaitAdvance();
    System.out.println("Стадия " + curPhase + " завершена");

    // Ожидать завершения всеми потоками второй стадии.
    curPhase = phsr.getPhase();
    phsr.arriveAndAwaitAdvance();
    System.out.println("Стадия " + curPhase + " завершена");

    curPhase = phsr.getPhase();
    phsr.arriveAndAwaitAdvance();
    System.out.println("Стадия " + curPhase + " завершена");

    // Отменить регистрацию главного потока.
    phsr.arriveAndDeregister();

    if(phsr.isTerminated())
      System.out.println("Объект Phaser закончил работу");
  }
}

// Поток выполнения, который использует объект Phaser.
class MyThread implements Runnable {
  Phaser phsr;
  String name;

  MyThread(Phaser p, String n) {
    phsr = p;
    name = n;
    phsr.register();
  }

  public void run() {
    System.out.println("Поток " + name + " начал первую стадию");
    phsr.arriveAndAwaitAdvance();  // Сигнализировать о прибытии.

    // Организовать небольшую паузу, чтобы предотвратить беспорядочный
    // вывод. Это делается только в целях иллюстрации и не требуется
    // для корректной работы объекта Phaser.
    try {
      Thread.sleep(100);
    } catch(InterruptedException e) {
      System.out.println(e);
    }

    System.out.println("Поток " + name + " начал вторую стадию");
    phsr.arriveAndAwaitAdvance();  // Сигнализировать о прибытии.
```

```
      // Организовать небольшую паузу, чтобы предотвратить беспорядочный
      // вывод. Это делается
      // только в целях иллюстрации и не требуется для корректной работы
      // объекта Phaser.
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.out.println(e);
      }

      System.out.println("Поток " + name + " начал третью стадию");
      phsr.arriveAndDeregister();  // Сигнализировать о прибытии
                                   // и отметить регистрацию.
    }
  }
```

Вот вывод их программы (у вас он может отличаться):

```
Начало
Поток A начал первую стадию
Поток C начал первую стадию
Поток B начал первую стадию
Стадия 0 завершена
Поток B начал вторую стадию
Поток C начал вторую стадию
Поток A начал вторую стадию
Стадия 1 завершена
Поток C начал третью стадию
Поток B начал третью стадию
Поток A начал третью стадию
Стадия 2 завершена
Объект Phaser закончил работу
```

Давайте внимательно взглянем на ключевые разделы программы. Первым делом в методе `main()` создается объект `Phaser` по имени `phsr` с начальным количеством сторон, равным 1 (что соответствует главному потоку). Далее запускаются три потока за счет создания трех объектов `MyThread`. Обратите внимание, что конструктору `MyThread` передается ссылка на `phsr` (объект `Phaser`). Объекты `MyThread` применяют этот объект `Phaser` для синхронизации своих действий. Затем в `main()` вызывается метод `getPhase()` для получения номера текущей стадии (изначально равного нулю) и метод `arriveAndAwaitAdvance()`. В результате выполнение метода `main()` приостанавливается до завершения нулевой стадии, что не случится до тех пор, пока все объекты `MyThread` тоже не вызовут `arriveAndAwaitAdvance()`. Когда это происходит, выполнение метода `main()` возобновляется, выводится сообщение о завершении нулевой стадии и происходит переход к следующей стадии. Процесс повторяется до тех пор, пока не будут завершены все три стадии. Затем в `main()` вызывается метод `arriveAndDeregister()`. В данный момент все три объекта `MyThread` также отменили регистрацию. Поскольку при переходе объекта `Phaser` к следующей стадии оказывается, что зарегистрированных сторон больше нет, объект `Phaser` заканчивает работу.

Теперь взгляните на класс MyThread. Прежде всего, обратите внимание, что конструктору передается ссылка на объект Phaser, который он будет использовать, после чего он регистрируется в новом потоке как сторона этого объекта. Таким образом, каждый новый экземпляр MyThread становится стороной, зарегистрированной в переданном объекте Phaser. Кроме того, каждый поток имеет три стадии. В приведенном примере каждая стадия состоит из заполнителя, который просто отображает имя потока и сведения о том, что он делает. Очевидно, что в реальном коде поток будет выполнять более значимые действия. Между первыми двумя стадиями поток вызывает метод arriveAndAwaitAdvance(). В итоге каждый поток ожидает до тех пор, пока все потоки не завершат стадию (и главный поток не будет готов). После прибытия всех потоков (включая главный поток) объект Phaser переходит к следующей стадии. После третьей стадии каждый поток отменяет свою регистрацию с помощью вызова arriveAndDeregister(). Как поясняется в комментариях внутри кода MyThread, вызовы метода sleep() предназначены для иллюстративных целей, чтобы предотвратить перемешивание вывода из-за многопоточности. Для корректной работы объекта Phaser они не нужны. Если их удалить, то вывод может выглядеть немного беспорядочным, но стадии все равно будут правильно синхронизироваться.

И еще один момент: хотя в предыдущем примере применялись три потока одного типа, это не является обязательным требованием. Каждая сторона, использующая объект Phaser, может быть уникальной и выполнять какую-то обособленную задачу. То, что происходит при переходе к следующей стадии, поддается контролю. Понадобится лишь переопределить метод onAdvance(), который вызывается во время выполнения, когда объект Phaser совершает переход от одной стадии к другой:

```
protected boolean onAdvance(int phase, int numParties)
```

В phase будет содержаться номер текущей стадии до инкрементирования, а в numParties — количество зарегистрированных сторон. Метод onAdvance() должен возвратить true, чтобы закончить работу объекта Phaser, или false, чтобы он продолжил работать. Стандартная версия onAdvance() возвращает true (т.е. завершает работу объекта Phaser), когда отсутствуют зарегистрированные стороны. Как правило, при переопределении тоже необходимо следовать такой практике.

Одна из причин переопределения метода onAdvance() заключается в том, чтобы позволить объекту Phaser выполнить определенное количество стадий и остановиться. Показанный ниже пример даст представление о применении подобного рода. В нем создается класс по имени MyPhaser, который расширяет Phaser, так что он будет выполнять указанное количество стадий. Цель достигается переопределением метода onAdvance(). Конструктор MyPhaser принимает один аргумент с количеством выполняемых стадий. Обратите внимание, что MyPhaser автоматически регистрирует одну сторону. Такое поведение полезно в данном примере, но потребности других приложений могут отличаться.

```
// Пример расширения класса Phaser и переопределения метода onAdvance(),
// чтобы выполнялось только заданное количество стадий.

import java.util.concurrent.*;

// Расширить класс MyPhaser, чтобы разрешить выполнение
// только указанного количества стадий.
class MyPhaser extends Phaser {
  int numPhases;

  MyPhaser(int parties, int phaseCount) {
    super(parties);
    numPhases = phaseCount - 1;
  }

  // Переопределить метод onAdvance() для выполнения указанного
  // количества стадий.
  protected boolean onAdvance(int p, int regParties) {
    // Этот оператор println() предназначен только для иллюстративных целей
    // Обычно метод onAdvance() ничего не отображает.
    System.out.println("Стадия " + p + " завершена.\n");

    // Если все стадии завершены, тогда возвратить true.
    if(p == numPhases || regParties == 0) return true;

    // В противном случае возвратить false.
    return false;
  }
}

class PhaserDemo2 {
  public static void main(String[] args) {

    MyPhaser phsr = new MyPhaser(1, 4);

    System.out.println("Начало\n");

    new Thread(new MyThread(phsr, "A")).start();
    new Thread(new MyThread(phsr, "B")).start();
    new Thread(new MyThread(phsr, "C")).start();

    // Ожидать завершения указанного количества стадий.
    while(!phsr.isTerminated()) {
      phsr.arriveAndAwaitAdvance();
    }

    System.out.println("Объект Phaser закончил работу.");
  }
}

// Поток выполнения, который использует Phaser.
class MyThread implements Runnable {
  Phaser phsr;
  String name;

  MyThread(Phaser p, String n) {
    phsr = p;
    name = n;
    phsr.register();
  }
```

```
  public void run() {
    while(!phsr.isTerminated()) {
      System.out.println("Поток " + name + " начинает стадию " +
                         phsr.getPhase());
      phsr.arriveAndAwaitAdvance();
      // Организовать небольшую паузу, чтобы предотвратить беспорядочный
      // вывод. Это делается только в целях иллюстрации и не требуется
      // для корректной работы объекта Phaser.
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.out.println(e);
      }
    }
  }
}
```

Вот вывод, генерируемый программой:

```
Начало

Поток B начинает стадию 0
Поток A начинает стадию 0
Поток C начинает стадию 0
Стадия 0 завершена.

Поток A начинает стадию 1
Поток B начинает стадию 1
Поток C начинает стадию 1
Стадия 1 завершена.

Поток C начинает стадию 2
Поток B начинает стадию 2
Поток A начинает стадию 2
Стадия 2 завершена.

Поток C начинает стадию 3
Поток B начинает стадию 3
Поток A начинает стадию 3
Стадия 3 завершена.

Объект Phaser закончил работу.
```

В методе `main()` создается один экземпляр класса `Phaser`, конструктору которого передается 4 в качестве аргумента, т.е. он выполнит четыре стадии и остановится. Далее создаются три потока, после чего организуется следующий цикл:

```
// Ожидать завершения указанного количества стадий.
while(!phsr.isTerminated()) {
  phsr.arriveAndAwaitAdvance();
}
```

В цикле просто вызывается метод `arriveAndAwaitAdvance()` до тех пор, пока объект `Phaser` не закончит свою работу, что произойдет, когда выполнится указанное количество стадий. В данном случае цикл продолжается,

пока не будут выполнены четыре стадии. Далее обратите внимание, что потоки тоже вызывают метод arriveAndAwaitAdvance() в цикле, который выполняется до тех пор, пока объект Phaser не прекратит работу, т.е. они будут выполняться вплоть до завершения указанного количества стадий.

Теперь внимательно взгляните на код метода onAdvance(). При каждом вызове методу onAdvance() передается текущая стадия и количество зарегистрированных сторон. Если текущая стадия соответствует указанной стадии или если количество зарегистрированных сторон равно нулю, то onAdvance() возвращает true, тем самым останавливая объект Phaser. Цель достигается с помощью такой строки кода:

```
// Если все стадии завершены, тогда возвратить true.
if(p == numPhases || regParties == 0) return true;
```

Как видите, для получения желаемого результата требуется очень мало кода.

Прежде чем двигаться дальше, полезно отметить, что для простого переопределения метода onAdvance() явно расширять класс Phaser, как в предыдущем примере, вовсе не обязательно. В ряде случаев можно создать более компактный код, используя для переопределения метода onAdvance() анонимный внутренний класс.

Класс Phaser обладает дополнительными возможностями, которые могут быть полезны в разрабатываемых приложениях. Вызвав метод awaitAdvance(), можно дождаться определенной стадии:

```
int awaitAdvance(int phase)
```

В phase указывается номер стадии, на которой метод awaitAdvance() будет ожидать перехода к следующей стадии. Он немедленно возвратит управление, если аргумент, переданный в phase, не соответствует текущей стадии. Кроме того, метод awaitAdvance() немедленно возвратит управление, когда объект Phaser прекращает работу. Тем не менее, если в phase передается текущая стадия, тогда awaitAdvance() будет ожидать инкрементирования стадии. Метод awaitAdvance() должен вызываться только зарегистрированной стороной. Существует также прерываемая версия этого метода, которая называется awaitAdvanceInterruptably().

Метод bulkRegister() предназначен для регистрации более одной стороны. Метод getRegisteredParties() позволяет получить количество зарегистрированных сторон. С помощью методов getArrivedParties() и getUnarrivedParties() можно получить количество прибывших и не прибывших сторон соответственно. Чтобы принудительно перевести объект Phaser в состояние окончания работы, необходимо вызвать метод forceTermination().

Допускается создавать дерево объектов Phaser, что поддерживается двумя дополнительными конструкторами, которые позволяют указать родителя, и методом getParent().

Использование исполнителя

Параллельный API предоставляет средство, называемое *исполнителем*, которое инициирует и управляет выполнением потоков. Следовательно, исполнитель предлагает альтернативу управлению потоками через класс Thread.

В основе исполнителя лежит интерфейс Executor, в котором определен один метод:

```
void execute(Runnable thread)
```

В thread передается поток, подлежащий выполнению. Таким образом, метод execute() запускает указанный поток.

Интерфейс ExecutorService расширяет Executor, добавляя методы, которые помогают управлять и контролировать выполнение потоков. Например, в ExecutorService определен метод shutdown(), останавливающий вызов ExecutorService:

```
void shutdown()
```

В интерфейсе ExecutorService также определены методы, которые выполняют потоки, возвращающие результаты, выполняют набор потоков и определяют состояние завершения работы; часть из них рассматривается позже в главе.

Вдобавок определен интерфейс ScheduledExecutorService, который расширяет ExecutorService для поддержки планирования потоков.

В параллельном API определены три готовых класса исполнителей: ThreadPoolExecutor, ScheduledThreadPoolExecutor и ForkJoinPool.

Класс ThreadPoolExecutor реализует интерфейсы Executor и ExecutorService и обеспечивает поддержку управляемого пула потоков. Класс ScheduledThreadPoolExecutor реализует также и интерфейс ScheduledExecutorService, позволяющий планировать пул потоков. Класс ForkJoinPool реализует интерфейсы Executor и ExecutorService и применяется инфраструктурой Fork/Join Framework, описанной далее в главе.

Пул потоков предоставляет набор потоков, которые используются для выполнения разнообразных задач. Вместо инициирования каждой задачей собственного потока применяются потоки из пула, что уменьшает накладные расходы, связанные с созданием множества отдельных потоков. Хотя классы ThreadPoolExecutor и ScheduledThreadPoolExecutor можно использовать напрямую, чаще всего исполнитель будет получаться вызовом один из статических фабричных методов, определенных в служебном классе Executors. Вот несколько примеров:

```
static ExecutorService newCachedThreadPool()
static ExecutorService newFixedThreadPool(int numThreads)
static ScheduledExecutorService newScheduledThreadPool(int numThreads)
```

Метод newCachedThreadPool() создает пул потоков, который по мере необходимости добавляет потоки, но если возможно, то повторно их использует. Метод newFixedThreadPool() создает пул потоков, состоящий из ука-

занного количества потоков. Метод newScheduledThreadPool() создает пул потоков, поддерживающий планирование потоков. Каждый метод возвращает ссылку на ExecutorService, которую можно применять для управления пулом.

Простой пример использования исполнителя

Прежде чем двигаться дальше, полезно ознакомиться с простым примером использования исполнителя. В приведенной ниже программе создается фиксированный пул потоков, содержащий два потока. Затем готовый пул применяется для выполнения четырех задач. Таким образом, четыре задачи совместно используют два потока, находящихся в пуле. После завершения задач пул закрывается и программа завершается.

```
// Простой пример применения исполнителя.

import java.util.concurrent.*;

class SimpExec {
  public static void main(String[] args) {
    CountDownLatch cdl = new CountDownLatch(5);
    CountDownLatch cdl2 = new CountDownLatch(5);
    CountDownLatch cdl3 = new CountDownLatch(5);
    CountDownLatch cdl4 = new CountDownLatch(5);
    ExecutorService es = Executors.newFixedThreadPool(2);

    System.out.println("Начало");

    // Запустить потоки.
    es.execute(new MyThread(cdl, "A"));
    es.execute(new MyThread(cdl2, "B"));
    es.execute(new MyThread(cdl3, "C"));
    es.execute(new MyThread(cdl4, "D"));

    try {
      cdl.await();
      cdl2.await();
      cdl3.await();
      cdl4.await();
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    es.shutdown();
    System.out.println("Конец");
  }
}

class MyThread implements Runnable {
  String name;
  CountDownLatch latch;

  MyThread(CountDownLatch c, String n) {
    latch = c;
    name = n;
  }
```

```
  public void run() {
    for(int i = 0; i < 5; i++) {
      System.out.println(name + ": " + i);
      latch.countDown();
    }
  }
}
```

Вот как выглядит вывод из программы (порядок выполнения потоков может варьироваться):

```
Начало
A: 0
A: 1
A: 2
A: 3
A: 4
C: 0
C: 1
C: 2
C: 3
C: 4
D: 0
D: 1
D: 2
D: 3
D: 4
B: 0
B: 1
B: 2
B: 3
B: 4
Конец
```

В выводе видно, что хотя в пуле потоков есть только два потока, все равно выполняются четыре задачи. Однако одновременно могут функционировать только две задачи, а остальные должны ждать, пока один из потоков в пуле не станет доступным для эксплуатации.

Вызов метода `shutdown()` очень важен. В его отсутствие программа не завершится, потому что исполнитель остается активным. Можете проверить, закомментировав вызов `shutdown()` и взглянув на результаты.

Использование интерфейсов `Callable` и `Future`

Одной из самых интересных средств параллельного API является интерфейс `Callable`, который представляет поток, возвращающий значение. Приложение может задействовать объекты `Callable` для вычисления результатов, которые затем возвращаются вызывающему потоку. Это мощный механизм, поскольку он облегчает кодирование многих типов численных расчетов, в которых частичные результаты вычисляются одновременно. Его также можно применять для запуска потока, который возвращает код состояния, указывающий на успешное завершение потока.

Callable — обобщенный интерфейс со следующим определением:

```
interface Callable<V>
```

В V указывается тип данных, возвращаемых задачей. В интерфейсе Callable определен единственный метод call():

```
V call() throws Exception
```

Внутри метода call() определяется задача, которую необходимо выполнить. После завершения этой задачи возвращается результат. Если результат не может быть вычислен, тогда метод call() обязан сгенерировать исключение.

Задача Callable выполняется реализацией ExecutorService путем вызова ее метода submit(). Существуют три формы submit(), но только одна из них используется для выполнения объекта Callable:

```
<T> Future<T> submit(Callable<T> task)
```

В task передается объект Callable, который будет выполняться в собственном потоке. Результат возвращается через объект типа Future.

Future — обобщенный интерфейс, представляющий значение, которое будет возвращено объектом реализации Callable. Поскольку значение получается в какой-то момент в будущем, имя Future (будущее) для него подходит как нельзя лучше. Интерфейс Future определен следующим образом:

```
interface Future<V>
```

В V указывается тип результата.

Чтобы получить возвращаемое значение, понадобится вызвать метод get() интерфейса Future, который имеет две формы:

```
V get()
  throws InterruptedException, ExecutionException
V get(long wait, TimeUnit tu)
  throws InterruptedException, ExecutionException, TimeoutException
```

Первая форма метода get() ожидает результат неопределенно долго. Вторая форма позволяет задать в wait период тайм-аута, который выражен в единицах, указанных в tu с помощью объекта перечисления TimeUnit (рассматривается позже в главе).

В приведенной далее программе иллюстрируется применение интерфейсов Callable и Future за счет создания трех задач, которые выполняют три разных расчета. Первая задача возвращает сумму значения, вторая рассчитывает длину гипотенузы прямоугольного треугольника по длинам его сторон, а третья вычисляет факториал значения. Все три расчета происходят одновременно.

```
// Пример использования Callable.

import java.util.concurrent.*;

class CallableDemo {
  public static void main(String[] args) {
    ExecutorService es = Executors.newFixedThreadPool(3);
```

```
    Future<Integer> f;
    Future<Double> f2;
    Future<Integer> f3;

    System.out.println("Начало");

    f = es.submit(new Sum(10));
    f2 = es.submit(new Hypot(3, 4));
    f3 = es.submit(new Factorial(5));

    try {
      System.out.println(f.get());
      System.out.println(f2.get());
      System.out.println(f3.get());
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
    catch (ExecutionException exc) {
      System.out.println(exc);
    }

    es.shutdown();
    System.out.println("Конец");
  }
}

// Далее следуют три вычислительных потока.
class Sum implements Callable<Integer> {
  int stop;

  Sum(int v) { stop = v; }

  public Integer call() {
    int sum = 0;
    for(int i = 1; i <= stop; i++) {
      sum += i;
    }
    return sum;
  }
}

class Hypot implements Callable<Double> {
  double side1, side2;

  Hypot(double s1, double s2) {
    side1 = s1;
    side2 = s2;
  }

  public Double call() {
    return Math.sqrt((side1*side1) + (side2*side2));
  }
}

class Factorial implements Callable<Integer> {
  int stop;

  Factorial(int v) { stop = v; }

  public Integer call() {
    int fact = 1;
```

```
    for(int i = 2; i <= stop; i++) {
      fact *= i;
    }
    return fact;
  }
}
```

Вот вывод из программы:

```
Начало
55
5.0
120
Конец
```

Перечисление TimeUnit

В параллельном API определено несколько методов, которые принимают аргумент типа TimeUnit, задающий период тайм-аута. Перечисление TimeUnit используется для указания *степени детализации* (или разрешения) синхронизации и определено в пакете java.util.concurrent. Ниже представлены его значения:

```
DAYS
HOURS
MINUTES
SECONDS
MICROSECONDS
MILLISECONDS
NANOSECONDS
```

Хотя в вызовах методов, принимающих параметр синхронизации, можно указывать любое значение перечисления TimeUnit, нет никакой гарантии, что система способна поддерживать запрошенное разрешение.

Рассмотрим пример, в котором применяется перечисление TimeUnit. Показанный в предыдущем разделе класс CallableDemo модифицирован, чтобы использовать вторую форму метода get(), принимающую аргумент TimeUnit.

```
try {
  System.out.println(f.get(10, TimeUnit.MILLISECONDS));
  System.out.println(f2.get(10, TimeUnit.MILLISECONDS));
  System.out.println(f3.get(10, TimeUnit.MILLISECONDS));
} catch (InterruptedException exc) {
  System.out.println(exc);
}
catch (ExecutionException exc) {
  System.out.println(exc);
} catch (TimeoutException exc) {
  System.out.println(exc);

}
```

В этой версии ни один вызов get() не будет ожидать более 10 миллисекунд.

В перечислении TimeUnit определены различные методы преобразования единиц измерения. Ниже приведены те, что изначально определены в TimeUnit:

```
long convert(long tval, TimeUnit tu)
long toMicros(long tval)
long toMillis(long tval)
long toNanos(long tval)
long toSeconds(long tval)
long toDays(long tval)
long toHours(long tval)
long toMinutes(long tval)
```

Метод convert() преобразует tval в указанные единицы измерения и возвращает результат. Методы toX() выполняют соответствующие преобразования и возвращают результат. В версии JDK 9 набор таких методов пополнился методами toChronoUnit() и of(), которые осуществляют преобразование между значениями java.time.temporal.ChronoUnit и TimeUnit. В версии JDK 11 появилась еще одна версия метода convert(), которая преобразует объект java.time.Duration в тип long.

В TimeUnit также определены следующие методы синхронизации:

```
void sleep(long delay) throws InterruptedExecution
void timedJoin(Thread thrd, long delay) throws InterruptedExecution
void timedWait(Object obj, long delay) throws InterruptedExecution
```

Метод sleep() приостанавливает выполнение на заданный период задержки, который указывается в виде константы перечисления вызывающего кода. Он транслируется в вызов Thread.sleep(). Метод timedJoin() является специализированной версией Thread.join(), в которой thrd делает паузу на период времени delay, заданный в единицах времени вызывающего кода. Метод timedWait() представляет собой специализированную версию Object.wait(), в которой obj ожидает в течение периода времени delay, указанного в единицах времени вызывающего кода.

Параллельные коллекции

Как уже объяснялось, в параллельном API определено несколько классов коллекций, разработанных для параллельной работы:

```
ArrayBlockingQueue
ConcurrentHashMap
ConcurrentLinkedDeque
ConcurrentLinkedQueue
ConcurrentSkipListMap
ConcurrentSkipListSet
CopyOnWriteArrayList
CopyOnWriteArraySet
DelayQueue
LinkedBlockingDeque
LinkedBlockingQueue
LinkedTransferQueue
PriorityBlockingQueue
SynchronousQueue
```

Они предлагают параллельные альтернативы соответствующим классам из Collections Framework и работают аналогично другим коллекциям, но обеспечивают поддержку параллелизма. У программистов, знакомых с Collections Framework, не должны возникать проблемы с применением параллельных коллекций.

Блокировки

Пакет `java.util.concurrent.locks` обеспечивает поддержку *блокировок*, которые представляют собой объекты, предлагающие альтернативу использованию ключевого слова `synchronized` для управления доступом к общему ресурсу. Давайте выясним, как работает блокировка. Перед доступом к общему ресурсу запрашивается блокировка, которая защищает этот ресурс. После окончания работы с ресурсом блокировка освобождается. Если второй поток попытается запросить блокировку, когда она задействована другим потоком, то второй поток будет приостановлен до тех пор, пока блокировка не будет освобождена. Подобным образом предотвращается конфликтующий доступ к общему ресурсу.

Блокировки особенно полезны, когда нескольким потокам необходимо получить доступ к значению из общих данных. Например, приложение инвентаризации может иметь поток, который сначала подтверждает наличие товара на складе, а затем по мере каждой продажи уменьшает количество имеющихся товаров. Если выполняются два или более таких потока, то без какой-либо формы синхронизации один поток может находиться в середине транзакции, когда второй поток начинает свою транзакцию. В результате оба потока будут предполагать, что существует достаточный запас, даже если имеется запас, достаточный только для удовлетворения одной продажи. В такой ситуации блокировка предлагает удобное средство обработки необходимой синхронизации.

Блокировка определяется с помощью интерфейса `Lock`, методы которого кратко описаны в табл. 29.2. Для получения блокировки необходимо вызвать метод `lock()`. Если блокировка недоступна, тогда `lock()` будет ожидать. Чтобы освободить блокировку, понадобится вызвать метод `unlock()`. Выяснить, доступна ли блокировка, и получить ее при условии доступности, можно посредством метода `tryLock()`.

Метод `tryLock()` не будет ожидать блокировку, когда она недоступна. Взамен он возвращает `true`, если блокировка получена, или `false` в противном случае. Метод `newCondition()` возвращает объект класса `Condition`, ассоциированный с блокировкой. Класс `Condition` обеспечивает полный контроль над блокировкой через такие методы, как `await()` и `signal()`, которые предоставляют функциональность сродни той, что предлагают методы `Object.wait()` и `Object.notify()`.

Таблица 29.2. Методы, определенные в интерфейсе Lock

Метод	Описание
`void lock()`	Ожидает до тех пор, пока вызываемая блокировка не будет получена
`void lockInterruptibly() throws InterruptedException`	Ожидает до тех пор, пока вызываемая блокировка не будет получена, при условии, что не произошло прерывание
`Condition newCondition()`	Возвращает объект `Condition`, ассоциированный с вызываемой блокировкой
`boolean tryLock()`	Пытается получить блокировку. Этот метод не будет ожидать, когда блокировка недоступна. Взамен он возвращает `true`, если блокировка была получена, или `false`, если блокировка в текущий момент используется другим потоком
`boolean tryLock (long wait, TimeUnit tu) throws InterruptedException`	Пытается получить блокировку. Если блокировка недоступна, тогда этот метод будет ожидать в течение промежутка времени, который указан в `wait` и выражен в единицах `tu`. Метод возвращает `true`, если блокировка была получена, или `false`, если блокировка не может быть получена в пределах указанного промежутка времени
`void unlock()`	Освобождает блокировку

В пакете `java.util.concurrent.locks` имеется реализация интерфейса `Lock` по имени `ReentrantLock`. Класс `ReentrantLock` определяет ***реентерабельную блокировку***, т.е. блокировку, в которую может многократно входить поток, в текущий момент удерживающий блокировку. Разумеется, в случае повторного входа потока в блокировку все вызовы `lock()` должны компенсироваться равным количеством вызовов `unlock()`, иначе поток, пытающийся получить блокировку, будет приостановлен до тех пор, пока блокировка не перестанет использоваться.

Применение блокировки демонстрируется в показанной ниже программе. В ней создаются два потока, которые обращаются к общему ресурсу по имени `Shared.count`. Прежде чем поток сможет иметь доступ к `Shared.count`, он должен получить блокировку. После получения блокировки значение `Shared.count` инкрементируется, а перед снятием блокировки поток приостанавливается, что вынуждает второй поток попытаться получить блокировку. Тем не менее, поскольку блокировка по-прежнему удерживается пер-

вым потоком, второй поток должен ожидать, пока первый поток не выйдет из состояния приостановки и освободит блокировку. Вывод показывает, что доступ к Shared.count действительно синхронизируется блокировкой.

```
// Простой пример использования блокировки.
import java.util.concurrent.locks.*;
class LockDemo {
  public static void main(String[] args) {
    ReentrantLock lock = new ReentrantLock();
    new Thread(new LockThread(lock, "A")).start();
    new Thread(new LockThread(lock, "B")).start();
  }
}
// Общий ресурс.
class Shared {
  static int count = 0;
}
// Поток, который инкрементирует count.
class LockThread implements Runnable {
  String name;
  ReentrantLock lock;
  LockThread(ReentrantLock lk, String n) {
    lock = lk;
    name = n;
  }
  public void run() {
    System.out.println("Начало " + name);
    try {
      // Заблокировать count.
      System.out.println(name + " ожидает блокировку count.");
      lock.lock();
      System.out.println(name + " блокирует count.");
      Shared.count++;
      System.out.println(name + ": " + Shared.count);
      // Разрешить переключение контекста, если это возможно.
      System.out.println(name + " в состоянии ожидания.");
      Thread.sleep(1000);
    } catch (InterruptedException exc) {
      System.out.println(exc);
    } finally {
      // Снять блокировку.
      System.out.println(name + " снимает блокировку count.");
      lock.unlock();
    }
  }
}
```

Вот вывод, генерируемый программой (порядок выполнения потоков может варьироваться):

```
Начало A
A ожидает блокировку count.
A блокирует count.
A: 1
A в состоянии ожидания.
Начало B
B is ожидает блокировку count.
A снимает блокировку count.
B блокирует count.
B: 2
B в состоянии ожидания.
B снимает блокировку count.
```

В пакете `java.util.concurrent.locks` также имеется интерфейс `ReadWriteLock`, определяющий блокировку, которая поддерживает отдельные блокировки для доступа по чтению и записи. Он позволяет предоставлять множество блокировок для средств чтения ресурса при условии, что в ресурс не производится запись. Интерфейс `ReadWriteLock` реализован классом `ReentrantReadWriteLock`.

На заметку! Существует специализированная блокировка в виде класса `StampedLock`, который не реализует интерфейсы `Lock` или `ReadWriteLock`. Однако класс `StampedLock` предлагает механизм, позволяющий задействовать его аспекты вроде `Lock` или `ReadWriteLock`.

Атомарные операции

Пакет `java.util.concurrent.atomic` является альтернативой другим средствам синхронизации при чтении или записи значений переменных определенных типов. В нем предлагаются методы, которые получают, устанавливают или сравнивают значение переменной в одной непрерывной (т.е. атомарной) операции. Таким образом, никакой блокировки или другого механизма синхронизации не требуется.

Атомарные операции выполняются с использованием классов, подобных `AtomicInteger` и `AtomicLong`, и методов вроде `get()`, `set()`, `compareAndSet()`, `decrementAndGet()` и `getAndSet()`, которые выполняют действия, отраженные в их именах (т.е. получение, установка, сравнение и установка, декрементирование и получение, получение и установка).

В приведенном далее примере демонстрируется синхронизация доступа к общему целому числу с применением `AtomicInteger`:

```
// Простой пример использования атомарных операций.
import java.util.concurrent.atomic.*;
class AtomicDemo {
  public static void main(String[] args) {
    new Thread(new AtomThread("A")).start();
    new Thread(new AtomThread("B")).start();
    new Thread(new AtomThread("C")).start();
  }
}
```

```
class Shared {
  static AtomicInteger ai = new AtomicInteger(0);
}
// Поток выполнения, который инкрементирует счетчик.
class AtomThread implements Runnable {
  String name;
  AtomThread(String n) {
    name = n;
  }
  public void run() {
    System.out.println("Начало " + name);
    for(int i=1; i <= 3; i++)
      System.out.println(name + " получено: " + Shared.ai.getAndSet(i));
  }
}
```

В классе Shared создается статическая переменная AtomicInteger по имени ai. Затем создаются три потока типа AtomThread. Внутри метода run() переменная Shared.ai модифицируется вызовом getAndSet(), который возвращает предыдущее значение и устанавливает значение, переданное в качестве аргумента. Использование AtomicInteger предотвращает одновременную запись в ai двумя потоками.

Таким образом, атомарные операции предлагают удобную (и, возможно, более эффективную) альтернативу другим механизмам синхронизации, когда задействована только одна переменная. Помимо других функциональных средств пакет java.util.concurrent.atomic предоставляет четыре класса, которые поддерживают накопительные операции, свободные от блокировок: DoubleAccumulator, DoubleAdder, LongAccumulator и LongAdder. Накопительные классы обеспечивают несколько определяемых пользователем операций, а суммирующие классы — нарастающий итог.

Параллельное программирование с помощью Fork/Join Framework

В последние годы в области разработки программного обеспечения сформировалась важная тенденция: *параллельное программирование*. Параллельное программирование, как правило, относится к методикам, которые пользуются преимуществами компьютеров с двумя и большим количеством процессоров (или ядер). Как известно большинству читателей, многоядерные компьютеры стали обычным явлением. Преимущество многопроцессорных сред связано с возможностью значительного повышения производительности программы. В результате возникла необходимость в механизме, который предложил бы программистам на Java простой, но эффективный способ эксплуатации нескольких процессоров ясным и масштабируемым образом. Для удовлетворения такой потребности в версии JDK 7 параллельный API получил несколько новых классов и интерфейсов, поддерживающих параллельное программирование, которые обычно называют *Fork/Join Framework*. Инфраструктура Fork/Join Framework находится в пакете java.util.concurrent.

Инфраструктура Fork/Join Framework улучшает многопоточное программирование в двух важных отношениях. Во-первых, она упрощает создание и использование множества потоков. Во-вторых, она автоматически задействует несколько процессоров. Другими словами, за счет применения Fork/Join Framework у приложений появляется возможность автоматически масштабироваться, чтобы использовать доступные процессоры. Указанные особенности делают инфраструктуру Fork/Join Framework рекомендуемым подходом к многопоточности, когда желательна параллельная обработка.

Прежде чем продолжить, важно отметить отличие между традиционной многопоточностью и параллельным программированием. В прошлом большинство компьютеров имели один процессор, а многопоточность главным образом применялась для извлечения выгоды из времени простоя, например, когда программа ожидала ввода данных пользователем. При таком подходе один поток может выполняться, пока другой ожидает.

Другими словами, в системе с одним процессором многопоточность используется для того, чтобы две или более задач могли совместно эксплуатировать процессор. Данный вид многопоточности обычно поддерживается объектом типа `Thread` (как было описано в главе 11). Хотя многопоточность подобного рода всегда будет оставаться весьма полезной, она не была оптимизирована для ситуаций, когда доступны два или более процессоров (или ядер).

При наличии нескольких процессоров требуется многопоточность второго вида, которая поддерживает настоящее параллельное выполнение. С двумя или более процессорами части программы можно выполнять одновременно, причем каждая часть будет выполняться на своем процессоре. Указанный подход позволяет значительно ускорить выполнение некоторых видов операций, скажем, сортировки, преобразования или поиска в крупном массиве. Во многих случаях такие операции можно разбить на более мелкие части (каждая из которых воздействует на часть массива) и выполнять каждую часть на своем процессоре. Понятно, что выигрыш в эффективности может оказаться огромным. Проще говоря, параллельное программирование станет частью будущего почти каждого программиста, потому что оно предлагает способ значительного повышения производительности программ.

Главные классы Fork/Join Framework

Инфраструктура Fork/Join Framework входит в состав пакета `java.util.concurrent`. В основе Fork/Join Framework лежат четыре класса, кратко описанные в табл. 29.3.

Давайте выясним, как они связаны друг с другом. Класс `ForkJoinPool` управляет выполнением задач `ForkJoinTask`.

Абстрактный класс `ForkJoinTask` расширяется абстрактными классами `RecursiveAction` и `RecursiveTask`. Обычно для создания задачи в коде указанные классы расширяются. Перед детальным исследованием процесса полезно ознакомиться с ключевыми аспектами каждого класса.

Таблица 29.3. Главные классы, определенные в инфраструктуре Fork/Join Framework

Класс	Описание
`ForkJoinTask<V>`	Абстрактный класс, который определяет задачу
`ForkJoinPool`	Класс, который управляет выполнением задач `ForkJoinTask`
`RecursiveAction`	Подкласс `ForkJoinTask<V>` для задач, не возвращающих значения
`RecursiveTask<V>`	Подкласс `ForkJoinTask<V>` для задач, возвращающих значения

На заметку! Класс `CountedCompleter` тоже расширяет `ForkJoinTask`, но обсуждение `CountedCompleter` выходит за рамки настоящей книги.

ForkJoinTask<V>

Абстрактный класс `ForkJoinTask<V>` определяет задачу, которой может управлять объект `ForkJoinPool`. В параметре типа V указывается тип результата задачи. Класс `ForkJoinTask` отличается от `Thread` тем, что представляет легковесную абстракцию задачи, а не поток выполнения. Задачи `ForkJoinTask` выполняются потоками, управляемыми пулом потоков типа `ForkJoinPool`. Данный механизм позволяет управлять крупным числом задач с помощью небольшого количества фактических потоков. Таким образом, задачи `ForkJoinTask` очень эффективны по сравнению с потоками. В классе `ForkJoinTask` определено много методов, основными из которых являются `fork()` и `join()`:

```
final ForkJoinTask<V> fork()
final V join()
```

Метод `fork()` отправляет вызывающую задачу на асинхронное выполнение, а это значит, что поток, вызывающий `fork()`, продолжит работать. После того, как задача запланирована для выполнения, метод `fork()` возвращает `this`. До выхода JDK 8 метод `fork()` мог запускаться только из вычислительной части другой задачи `ForkJoinTask`, которая функционирует в пуле `ForkJoinPool`. (Вскоре вы узнаете, как создавать вычислительную часть задачи.) Тем не менее, что касается современных версий Java, если метод `fork()` не вызывается во время выполнения в `ForkJoinPool`, тогда автоматически применяется общий пул. Метод `join()` ожидает завершения задачи, для которой он вызывается, и возвращает результат задачи. Таким образом, с помощью `fork()` и `join()` можно запустить одну или несколько новых задач и затем дождаться их завершения.

В классе `ForkJoinTask` есть еще один важный метод — `invoke()`. Он объединяет операции `fork()` и `join()`, т.к. начинает задачу и ожидает ее завершения; метод возвращает результат вызывающей задачи:

```
final V invoke()
```

Метод invokeAll() позволяет инициировать более одной задачи одновременно. Вот две его формы:

```
static void invokeAll(ForkJoinTask<?> taskA, ForkJoinTask<?> taskB)
static void invokeAll(ForkJoinTask<?> ... taskList)
```

Первая форма invokeAll() выполняет задачи taskA и taskB, а вторая форма — все задачи, указанные в taskList. В обоих случаях вызывающий поток ожидает завершения всех указанных задач. Как и fork(), изначально методы invoke() и invokeAll() могли выполняться только из вычислительной части другой задачи ForkJoinTask, функционирующей в пуле ForkJoinPool. Появление общего пула в JDK 8 ослабило это требование.

RecursiveAction

Класс RecursiveAction является подклассом ForkJoinTask и инкапсулирует задачу, которая не возвращает результат. Как правило, класс RecursiveAction расширяется в коде для создания задачи с возвращаемым типом void. В RecursiveAction определены четыре метода, но обычно интерес представляет только один из них: абстрактный метод по имени calculate(). При расширении RecursiveAction с целью создания конкретного класса код, определяющий задачу, помещается внутрь calculate(). Метод calculate() представляет вычислительную часть задачи.

Так выглядит определение метода compute() в классе RecursiveAction:

```
protected abstract void compute()
```

Обратите внимание, что метод calculate() объявлен защищенным и абстрактным, т.е. он должен быть реализован подклассом (если только подкласс тоже не является абстрактным).

В общем случае класс RecursiveAction используется для реализации рекурсивной стратегии "разделяй и властвуй" в отношении задач, которые не возвращают результатов (см. раздел "Стратегия 'разделяй и властвуй'" далее в главе).

RecursiveTask<V>

Класс RecursiveTask<V> — еще один подкласс ForkJoinTask, который инкапсулирует задачу, возвращающую результат. Тип результата указывается в V. Обычно класс RecursiveTask<V> расширяется в коде для создания задачи, возвращающей значение. Как и в классе RecursiveAction, наибольший интерес вызывает его абстрактный метод calculate(), поскольку он олицетворяет собой вычислительную часть задачи. В случае создания конкретного класса, расширяющего RecursiveTask<V>, внутрь метода calculate() помещается код, который представляет задачу. Данный код также должен возвращать результат задачи.

Метод compute() определен в классе RecursiveTask<V> следующим образом:

```
protected abstract V compute()
```

Обратите внимание, что метод compute() объявлен защищенным и абстрактным, т.е. он должен быть реализован подклассом. В случае реализации он обязан возвращать результат задачи.

В общем случае класс RecursiveTask применяется для реализации рекурсивной стратегии "разделяй и властвуй" в отношении задач, которые возвращают результаты (см. раздел "Стратегия 'разделяй и властвуй'" далее в главе).

ForkJoinPool

Выполнение задач ForkJoinTask происходит в пуле ForkJoinPool, который также управляет их выполнением. Следовательно, чтобы выполнить задачу ForkJoinTask, необходимо иметь экземпляр ForkJoinPool. Существуют два способа получения объекта ForkJoinPool. Во-первых, его можно создать явно с помощью конструктора класса ForkJoinPool. Во-вторых, можно воспользоваться так называемым *общим пулом*. Общий пул (появившийся в версии JDK 8) — это статическая переменная типа ForkJoinPool, автоматически доступная для применения. Далее будут описаны все способы, начиная с создания пула вручную.

В классе ForkJoinPool определено несколько конструкторов; ниже показаны два конструктора, которые используются чаще других:

```
ForkJoinPool()
ForkJoinPool(int pLevel)
```

Первый конструктор создает стандартный пул, поддерживает уровень параллелизма, который соответствует количеству процессоров, доступных в системе. Второй конструктор позволяет указать уровень параллелизма. Его значение должно быть больше нуля и не выходить за лимит, установленный в реализации. Уровень параллелизма определяет количество потоков, которые могут выполняться одновременно. В результате уровень параллелизма фактически определяет количество задач, которые способны выполняться одновременно. (Конечно, количество задач, которые могут выполняться одновременно, не может превышать количество процессоров.) Важно понимать, что уровень параллелизма *не* ограничивает число задач, которыми способен управлять пул. Количество задач, которыми в состоянии управлять класс ForkJoinPool, значительно больше его уровня параллелизма. Кроме того, уровень параллелизма является лишь целью, но не гарантией.

После создания экземпляра ForkJoinPool задачу можно запустить несколькими способами. Первая начатая задача нередко считается главной. Часто главная задача начинает подзадачи, которые также управляются пулом. Один из распространенных способов начать выполнение главной задачи предусматривает вызов в пуле ForkJoinPool метода invoke():

```
<T> T invoke(ForkJoinTask<T> task)
```

Метод invoke() запускает задачу, указанную в task, и возвращает результат задачи, т.е. вызывающий код ожидает, пока invoke() не возвратит управление.

Чтобы запустить задачу, не дожидаясь ее завершения, можно применить метод execute(). Вот одна из его форм:

```
void execute(ForkJoinTask<?> task)
```

В данном случае задача task запускается, но вызывающий код не будет ожидать ее завершения, а продолжит выполняться асинхронно.

В современных версиях Java нет необходимости явно создавать ForkJoinPool, т.к. доступен общий пул. В общем случае, если не используется явно созданный пул, тогда автоматически будет применяться общий пул. Хотя это не всегда необходимо, можно получить ссылку на общий пул, вызвав метод commonPool(), который определен в ForkJoinPool:

```
static ForkJoinPool commonPool()
```

Метод commonPool() возвращает ссылку на общий пул, который обеспечивает стандартный уровень параллелизма. Его можно установить с помощью системного свойства. (Подробности ищите в документации по API.) Как правило, стандартный общий пул будет хорошим вариантом для многих приложений. Разумеется, всегда можно сконструировать собственный пул.

Существуют два основных способа запуска задачи с использованием общего пула. Во-первых, посредством вызова метода commonPool() можно получить ссылку на пул и затем применять ее для вызова invoke() или execute(), как только что было описано. Во-вторых, для задачи вне ее вычислительной части можно вызывать методы ForkJoinTask вроде fork() или invoke(). В таком случае будет автоматически использоваться общий пул. Другими словами, методы fork() и invoke() запустят задачу с применением общего пула, если задача еще не функционирует внутри ForkJoinPool.

Класс ForkJoinPool управляет выполнением своих потоков, используя подход, который называется *перехватом работы*. Каждый рабочий поток поддерживает очередь задач. Если очередь одного рабочего потока пуста, тогда он возьмет задачу из другого рабочего потока, что увеличивает общую эффективность и помогает поддерживать сбалансированную нагрузку. (Из-за того, что процессорное время требуется другим процессам в системе, даже два рабочих потока с идентичными задачами в соответствующих очередях могут не завершиться одновременно.)

И еще один момент: в пуле ForkJoinPool применяются потоки демонов. Поток демона автоматически завершается, когда завершаются все пользовательские потоки. Таким образом, нет необходимости явно закрывать ForkJoinPool. Однако за исключением общего пула это можно делать, вызвав метод shutdown(), который не оказывает влияния на общий пул.

Стратегия "разделяй и властвуй"

Как правило, при работе с инфраструктурой Fork/Join Framework используется *стратегия "разделяй и властвуй"*, основанная на рекурсии. Именно потому два подкласса ForkJoinTask называются RecursiveAction и RecursiveTask. Ожидается, что один из указанных классов будет расширен для создания собственной задачи Fork/Join Framework.

Стратегия "разделяй и властвуй" базируется на рекурсивном разделении задачи на более мелкие подзадачи до тех пор, пока размер подзадачи не станет достаточно малым для последовательной обработки. Например, задача, применяющая преобразование к элементам массива N целых чисел, может быть разбита на две подзадачи, каждая из которых преобразует половину элементов в массиве. То есть одна подзадача преобразует элементы с 0 до $N/2$, а другая — элементы с $N/2$ до N. В свою очередь все подзадачи могут быть сведены к набору подзадач, каждая из которых преобразует половину оставшихся элементов. Такой процесс разделения массива продолжается до тех пор, пока не будет достигнут порог, при котором последовательное решение будет быстрее, нежели создание еще одного разделения.

Преимущество стратегии "разделяй и властвуй" заключается в том, что обработка может происходить параллельно. В итоге вместо того, чтобы циклически проходить через весь массив с использованием одного потока, фрагменты массива могут быть обработаны одновременно. Разумеется, стратегия "разделяй и властвуй" подходит во многих ситуациях, когда нет массива (или коллекции), но наиболее распространенное применение предусматривает наличие массива, коллекции или группы данных некоторого вида.

Одним из условий наилучшего использования стратегии "разделяй и властвуй" является корректный выбор порога, при котором применяется последовательная обработка (а не дальнейшее разделение). Оптимальный порог обычно получается в результате профилирования характеристик выполнения. Тем не менее, даже при использовании порога ниже оптимального все равно будет наблюдаться значительное ускорение. Однако лучше всего избегать чрезмерно больших или крайне малых порогов. На момент написания книги в документации по Java API для класса `ForkJoinTask<T>` приводится эмпирическое правило, в котором утверждается, что задача должна выполняться где-то за 100–10 000 вычислительных шагов.

Кроме того, важно понимать, что на оптимальное пороговое значение также влияет то, сколько времени занимает вычисление. Если каждый вычислительный шаг достаточно продолжителен, тогда меньшие пороговые значения могут оказаться лучше. И наоборот, если каждый вычислительный шаг довольно короткий, то лучшие результаты могут обеспечить более высокие пороговые значения. Для приложений, которые должны запускаться в известной системе с известным количеством процессоров, принять обоснованное решение относительно порогового значения можно на основе количества процессоров. С другой стороны, для приложений, которые будут функционировать в системах с заранее неизвестными возможностями, никаких предположений о среде выполнения выдвинуть не удастся.

И еще один момент: несмотря на то, что в системе может быть доступно несколько процессоров, другие задачи (и сама операционная система) будут соперничать с вашим приложением за процессорное время. Таким образом, важно не предполагать, что ваша программа получит неограниченный доступ ко всем процессорам. Кроме того, разные запуски одной и той же программы могут показывать отличающиеся характеристики времени выполнения из-за варьирующейся рабочей нагрузки.

Простой пример использования Fork/Join Framework

На данном этапе полезно рассмотреть простой пример, который демонстрирует удобство применения инфраструктуры Fork/Join Framework и стратегии "разделяй и властвуй". Ниже приведена программа, в которой элементы в массиве значений double преобразуются в их квадратные корни, что делается посредством подкласса RecursiveAction. Обратите внимание, что в программе создается собственный пул ForkJoinPool.

```
// Простой пример применения базовой стратегии "разделяй и властвуй".
// В этом случае используется RecursiveAction.
import java.util.concurrent.*;
import java.util.*;

// Задача ForkJoinTask, которая (через RecursiveAction) трансформирует
// элементы массива значений double в их квадратные корни.
class SqrtTransform extends RecursiveAction {
  // Произвольно установить пороговое значение в этом примере в 1000.
  // В реальном коде оптимальное пороговое значение может быть
  // выяснено за счет профилирования и экспериментирования.
  final int seqThreshold = 1000;

  // Массив, в который будет осуществляться доступ.
  double[] data;

  // Определить, какую часть данных обрабатывать.
  int start, end;

  SqrtTransform(double[] vals, int s, int e) {
    data = vals;
    start = s;
    end = e;
  }
  // Метод, в котором будут происходить параллельные вычисления.
  protected void compute() {

    // Если количество элементов меньше порогового значения,
    // тогда обрабатывать последовательно.
    if((end - start) < seqThreshold) {
      // Трансформировать каждый элемент в его квадратный корень.
      for(int i = start; i < end; i++) {
        data[i] = Math.sqrt(data[i]);
      }
    }
    else {
      // В противном случае продолжить разделение данных на меньшие части.

      // Найти среднюю точку.
      int middle = (start + end) / 2;

      // Запустить новые задачи, используя дополнительно разделенные
      // на части данные.
      invokeAll(new SqrtTransform(data, start, middle),
                new SqrtTransform(data, middle, end));
    }
  }
}
```

```
// Демонстрация параллельного выполнения.
class ForkJoinDemo {
  public static void main(String[] args) {
    // Создать пул задач.
    ForkJoinPool fjp = new ForkJoinPool();

    double[] nums = new double[100000];

    // Присвоить nums ряд значений.
    for(int i = 0; i < nums.length; i++)
      nums[i] = (double) i;

    System.out.println("Часть исходной последовательности:");

    for(int i=0; i < 10; i++)
      System.out.print(nums[i] + " ");
    System.out.println("\n");

    SqrtTransform task = new SqrtTransform(nums, 0, nums.length);

    // Запустить главную задачу ForkJoinTask.
    fjp.invoke(task);

    System.out.println("Часть трансформированной последовательности" +
                       " (с четырьмя знаками после десятичной точки):");
    for(int i=0; i < 10; i++)
      System.out.format("%.4f ", nums[i]);
    System.out.println();
  }
}
```

Вот вывод, сгенерированный программой:

```
Часть исходной последовательности:
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

Часть трансформированной последовательности (с четырьмя знаками после
десятичной точки):
0.0000 1.0000 1.4142 1.7321 2.0000 2.2361 2.4495 2.6458 2.8284 3.0000
```

Легко заметить, что значения элементов массива были трансформированы в их квадратные корни.

Давайте внимательно посмотрим, как работает программа. Прежде всего, обратите внимание, что класс `SqrtTransform` расширяет `RecursiveAction`. Как уже объяснялось, класс `RecursiveAction` расширяет `ForkJoinTask` для задач, которые не возвращают результатов. Теперь взгляните на финальную переменную `seqThreshold`. Ее значение определяет, когда будет начинаться последовательная обработка. Значение `seqThreshold` установлено (произвольно) в 1000. Далее обратите внимание, что ссылка на обрабатываемый массив хранится в `data`, а поля `start` и `end` служат для указания границ элементов, к которым нужно получить доступ.

Главное действие программы происходит в методе `compute()`. Он начинается с проверки того, что количество обрабатываемых элементов ниже порога последовательной обработки. Если это так, то элементы обрабатываются (в рассматриваемом примере путем вычисления их квадратного корня). Если порог последовательной обработки не достигнут, тогда с помощью вызова

invokeAll() запускаются две новые задачи, каждая из которых обрабатывает половину элементов. Как упоминалось ранее, метод invokeAll() ожидает до тех пор, пока не возвратится управление из обеих задач. После разворачивания всех рекурсивных вызовов каждый элемент массива окажется модифицированным, причем большая часть действий выполняется параллельно (если доступно несколько процессоров).

Выше отмечалось, что в настоящее время нет необходимости явно создавать ForkJoinPool, поскольку доступен общий пул. Вдобавок использовать общий пул очень просто. Скажем, получить ссылку на общий пул можно вызовом статического метода commonPool(), определенного в ForkJoinPool. Следовательно, предыдущую программу можно переписать для применения общего пула, заменив вызов конструктора ForkJoinPool вызовом commonPool():

```
ForkJoinPool fjp = ForkJoinPool.commonPool();
```

Впрочем, нет никакой необходимости явно получать ссылку на общий пул, т.к. вызов метода invoke() или fork() класса ForkJoinTask на задаче, которая еще не является частью пула, приведет к тому, что она автоматически будет выполняться в общем пуле. Например, в предыдущей программе можно полностью избавиться от переменной fjp и запускать задачу с помощью такой строки:

```
task.invoke();
```

Как видно из приведенного обсуждения, использовать общий пул может быть проще, чем создавать собственный пул. Кроме того, во многих случаях общий пул является предпочтительным подходом.

Влияние уровня параллелизма

Прежде чем двигаться дальше, важно понять влияние уровня параллелизма на производительность задачи Fork/Join Framework и то, как взаимодействуют параллелизм и порог. Программа, показанная в текущем разделе, позволяет экспериментировать с различными степенями параллелизма и пороговыми значениями. Предполагая запуск программы на многоядерном компьютере, можно в интерактивном режиме наблюдать за эффектом таких значений.

В предыдущем примере применялся стандартный уровень параллелизма, но задать желаемый уровень параллелизма несложно. Один из способов предусматривает его указание при создании экземпляра класса ForkJoinPool с помощью следующего конструктора:

```
ForkJoinPool(int pLevel)
```

В pLevel передается уровень параллелизма, который должен быть больше нуля и меньше ограничения, установленного реализацией.

В приведенной ниже программе создается задача Fork/Join Framework, которая трансформирует массив значений double. Трансформация произ-

вольна, но рассчитана на потребление нескольких циклов процессора. Так было сделано для того, чтобы более четко отражались эффекты изменения порогового значения или уровня параллелизма. При запуске программы понадобится указать в командной строке пороговое значение и уровень параллелизма. Затем программа запускает задачи. Она также отображает количество времени, необходимое для выполнения задач, с использованием метода `System.nanoTime()`, который возвращает значение таймера высокого разрешения машины JVM.

```
// Простая программа, которая позволяет экспериментировать с эффектами
// от изменения порогового значения и уровня параллелизма задачи ForkJoinTask

import java.util.concurrent.*;

// Задача ForkJoinTask, которая (через RecursiveAction)
// трансформирует элементы массива значений double.
class Transform extends RecursiveAction {

  // Порог последовательной обработки, который устанавливается конструктором
  int seqThreshold;

  // Массив, в который будет осуществляться доступ.
  double[] data;

  // Определить, какую часть данных обрабатывать.
  int start, end;

  Transform(double[] vals, int s, int e, int t ) {
    data = vals;
    start = s;
    end = e;
    seqThreshold = t;
  }

  // Метод, в котором будут происходить параллельные вычисления.
  protected void compute() {

    // Если количество элементов меньше порогового значения,
    // тогда обрабатывать последовательно.
    if((end - start) < seqThreshold) {
      // В следующем коде элементу с четным индексом присваивается
      // квадратный корень из его исходного значения. Элементу с нечетным
      // индексом присваивается кубический корень его исходного значения.
      // Этот код предназначен для простого потребления процессорного
      // времени, чтобы эффекты параллельного выполнения стали более заметными
      for(int i = start; i < end; i++) {
        if((data[i] % 2) == 0)
          data[i] = Math.sqrt(data[i]);
        else
          data[i] = Math.cbrt(data[i]);
      }
    }
    else {
      // В противном случае продолжить разделение данных на меньшие части.

      // Найти среднюю точку.
      int middle = (start + end) / 2;
```

```
      // Запустить новые задачи, используя дополнительно разделенные
      // на части данные.
      invokeAll(new Transform(data, start, middle, seqThreshold),
                new Transform(data, middle, end, seqThreshold));
    }
  }
}

// Демонстрация параллельного выполнения.
class FJExperiment {

  public static void main(String[] args) {
    int pLevel;
    int threshold;

    if(args.length != 2) {
      System.out.println("Использование: FJExperiment уровень-
параллелизма пороговое-значение");
      return;
    }

    pLevel = Integer.parseInt(args[0]);
    threshold = Integer.parseInt(args[1]);

    // Переменные, используемые для измерения времени выполнения задачи.
    long beginT, endT;

    // Создать пул задач. Обратите внимание на установку уровня параллелизма
    ForkJoinPool fjp = new ForkJoinPool(pLevel);

    double[] nums = new double[1000000];

    for(int i = 0; i < nums.length; i++)
      nums[i] = (double) i;

    Transform task = new Transform(nums, 0, nums.length, threshold);

    // Начать измерение времени.
    beginT = System.nanoTime();

    // Запустить главную задачу ForkJoinTask.
    fjp.invoke(task);

    // Закончить измерение времени.
    endT = System.nanoTime();

    System.out.println("Уровень параллелизма: " + pLevel);
    System.out.println("Порог последовательной обработки: " + threshold);
    System.out.println("Общее затраченное время: " + (endT - beginT) + " нс");
    System.out.println();
  }
}
```

При запуске программы необходимо указать уровень параллелизма и пороговое значение. Обязательно поэкспериментируйте с разными значениями для них и понаблюдайте за результатами. Помните, чтобы добиться эффективности, вы должны запускать код на компьютере, который имеет минимум два процессора. Кроме того, следует понимать, что два разных запуска могут (почти наверняка) выдавать разные результаты из-за того, что другие процессы в системе потребляют процессорное время.

Следующий эксперимент дает представление о разнице, которую обеспечивает параллелизм. Сначала запустите программу так:

```
java FJExperiment 1 1000
```

Здесь запрашивается уровень параллелизма 1 (по существу последовательное выполнение) с порогом 1000. Вот результат запуска, полученный на компьютере с двухъядерным процессором:

```
Уровень параллелизма: 1
Порог последовательной обработки: 1000
Общее затраченное время: 259677487 нс
```

А теперь укажите уровень параллелизма 2:

```
java FJExperiment 2 1000
```

Ниже показан пример вывода на том же компьютере с двухъядерным процессором:

```
Уровень параллелизма: 2
Порог последовательной обработки: 1000
Общее затраченное время: 169254472 нс
```

Как видите, добавление параллелизма существенно сокращает время выполнения, тем самым увеличивая скорость работы программы. Проведите эксперименты с изменением порогового значения и уровня параллелизма на своем компьютере. Результаты наверняка вас удивят.

Существуют еще два метода, которые могут оказаться полезными при экспериментировании с характеристиками выполнения программы Fork/Join Framework. Во-первых, можно получить уровень параллелизма, вызвав метод `getParallelism()`, который определен в `ForkJoinPool`:

```
int getParallelism()
```

Он возвращает текущий уровень параллелизма. Вспомните, что для создаваемых вами пулов это значение по умолчанию будет равно количеству доступных процессоров. (Чтобы получить уровень параллелизма для общего пула, также можно применять метод `getCommonPoolParallelism()`.) Во-вторых, можно получить количество процессоров, доступных в системе, с помощью вызова метода `availableProcessors()`, который определен в классе `Runtime`:

```
int availableProcessors()
```

Возвращаемое значение между вызовами может изменяться из-за других потребностей системы.

Пример использования `RecursiveTask<V>`

Предшествующие два примера основаны на `RecursiveAction`, т.е. они одновременно выполняют задачи, которые не возвращают результатов. Для создания задачи, возвращающей результат, нужно использовать класс `RecursiveTask`. В общем случае решения разрабатываются в той же мане-

ре, как только что было показано. Ключевое отличие состоит в том, что метод `calculate()` возвращает результат. Таким образом, результаты придется объединять в одно целое, чтобы после завершения первого вызова возвращался общий результат. Еще одно отличие связано с тем, что подзадача обычно запускается явным вызовом методов `fork()` и `join()`, а не неявно, вызывая, например, `invokeAll()`.

Применение класса `RecursiveTask` демонстрируется в следующей программе, где создается задача по имени `Sum`, которая возвращает сумму значений в массиве типа `double`. В этом примере массив состоит из 5000 элементов, причем значение каждого второго элемента является отрицательным. Таким образом, значения в начале массива выглядят как 0, –1, 2, –3, 4 и т.д. (Обратите внимание, что в примере создается собственный пул. В качестве упражнения можете попробовать модифицировать его, чтобы использовать общий пул.)

```
// Простой пример использования RecursiveTask<V>.

import java.util.concurrent.*;

// Задача RecursiveTask, которая вычисляет сумму значений
// в массиве типа double.
class Sum extends RecursiveTask<Double> {

  // Порог последовательной обработки.
  final int seqThresHold = 500;

  // Массив, в который будет осуществляться доступ.
  double[] data;

  // Определить, какую часть данных обрабатывать.
  int start, end;

  Sum(double[] vals, int s, int e ) {
    data = vals;
    start = s;
    end = e;
  }

  // Найти сумму значений в массиве типа double.
  protected Double compute() {
    double sum = 0;

    // Если количество элементов меньше порогового значения,
    // тогда обрабатывать последовательно.
    if((end - start) < seqThresHold) {
      // Суммировать элементы.
      for(int i = start; i < end; i++)
        sum += data[i];
    }
    else {
      // В противном случае продолжить разделение данных на меньшие части.

      // Найти среднюю точку.
      int middle = (start + end) / 2;

      // Запустить новые задачи, используя дополнительно разделенные
      // на части данные.
```

```
      Sum subTaskA = new Sum(data, start, middle);
      Sum subTaskB = new Sum(data, middle, end);

      // Запустить все задачи с помощью fork().
      subTaskA.fork();
      subTaskB.fork();

      // Ожидать, пока подзадачи возвратят управление,
      // и объединить результаты.
      sum = subTaskA.join() + subTaskB.join();
    }
    // Возвратить окончательную сумму.
    return sum;
  }
}
// Демонстрация параллельного выполнения.
class RecurTaskDemo {
  public static void main(String[] args) {
    // Создать пул задач.
    ForkJoinPool fjp = new ForkJoinPool();
    double[] nums = new double[5000];

    // Инициализировать nums чередующимися положительными
    // и отрицательными значениями.
    for(int i=0; i < nums.length; i++)
      nums[i] = (double) (((i%2) == 0) ? i : -i);
    Sum task = new Sum(nums, 0, nums.length);

    // Запустить задачи ForkJoinTask. Обратите внимание,
    // что в этом случае метод invoke() возвращает результат.
    double summation = fjp.invoke(task);

    System.out.println("Результат суммирования: " + summation);
  }
}
```

Вот вывод, сгенерированный программой:

```
Результат суммирования: -2500.0
```

В программе есть пара интересных особенностей. Прежде всего, обратите внимание, что две подзадачи выполняются путем вызова `fork()`:

```
subTaskA.fork();
subTaskB.fork();
```

В данном случае применяется метод `fork()`, потому что он запускает задачу, но не ожидает ее завершения. (Таким образом, задача выполняется асинхронно.) Результат каждой задачи получается вызовом метода `join()`:

```
sum = subTaskA.join() + subTaskB.join();
```

Приведенный выше оператор ожидает окончания каждой задачи. Затем он суммирует результаты из всех задач и присваивает итог переменной `sum`. Следовательно, сумма каждой подзадачи добавляется к промежуточному итогу. Наконец, метод `calculate()` завершается возвращением переменной `sum`, которая будет содержать окончательную сумму, когда возвратится управление из первого вызова.

Существуют и другие способы обеспечения асинхронного выполнения подзадач. Скажем, в показанном ниже коде метод `fork()` используется для запуска `subTaskA`, а метод `invoke()` — для запуска и ожидания `subTaskB`:

```
subTaskA.fork();
sum = subTaskB.invoke() + subTaskA.join();
```

Еще одна альтернатива заключается в том, чтобы подзадача `subTaskB` вызывала метод `compute()` напрямую:

```
subTaskA.fork();
sum = subTaskB.compute() + subTaskA.join();
```

Выполнение задачи асинхронным образом

В предшествующих программах для инициирования задачи вызывался метод `invoke()` на объекте `ForkJoinPool`. Такой подход обычно применяется, когда вызывающий поток должен дождаться завершения задачи (что часто бывает), поскольку метод `invoke()` не возвращает значение до тех пор, пока задача не будет окончена. Тем не менее, задачу можно запустить асинхронно. При таком подходе вызывающий поток продолжит выполняться. В итоге и вызывающий поток, и задача выполняются одновременно. Для запуска задачи асинхронным образом используется метод `execute()`, который тоже определен в `ForkJoinPool` и имеет две формы:

```
void execute(ForkJoinTask<?> task)
void execute(Runnable task)
```

В обеих формах `task` задает задачу, подлежащую запуску. Обратите внимание, что вторая форма метода `execute()` позволяет указать задачу `Runnable`, а не `ForkJoinTask`. Тем самым образуется мост между традиционным подходом Java к многопоточности и инфраструктурой Fork/Join Framework. Важно помнить, что потоки, применяемые `ForkJoinPool`, являются демонами, т.е. они закончатся тогда, когда закончится главный поток. В результате может потребоваться поддерживать главный поток в рабочем состоянии до тех пор, пока задачи не будут завершены.

Отмена задачи

Задачу можно отменить, вызвав метод `cancel()`, который определен в `ForkJoinTask`. Он имеет следующую общую форму:

```
boolean cancel(boolean interuptOK)
```

Метод `cancel()` возвращает `true`, если задача, на которой он был вызван, была отменена, либо `false`, если задача завершена или не может быть отменена. В настоящее время параметр `interruptOK` стандартной реализацией не используется. Как правило, метод `cancel()` предназначен для вызова из кода вне задачи, потому что задача может легко отменить себя за счет возвращения.

Выяснить, была ли задача отменена, можно с помощью метода `isCancelled()`:

```
final boolean isCancelled()
```

Он возвращает `true`, если вызывающая задача была отменена до своего завершения, или `false` в противном случае.

Определение состояния завершения задачи

Помимо только что описанного метода `isCancelled()` в классе `ForkJoinTask` определены еще два метода, которые можно применять для выяснения состояния завершения задачи. Первый из них — метод `isCompletedNormally()`:

```
final boolean isCompletedNormally()
```

Он возвращает `true`, если вызывающая задача завершилась нормально, т.е. не сгенерировала исключение и не была отменена посредством вызова `cancel()`. В противном случае метод `isCompletedNormally()` возвращает `false`. Ниже показан второй метод, `isCompletedAbnormally()`:

```
final boolean isCompletedAbnormally()
```

Метод `isCompletedAbnormally()` возвращает `true`, если вызывающая задача завершилась, поскольку она была отменена или сгенерировала исключение, либо `false` в противном случае.

Перезапуск задачи

Обычно повторно выполнить задачу невозможно. Другими словами, после завершения задачи перезапустить ее нельзя. Однако можно повторно инициализировать состояние задачи (после ее завершения), чтобы задачу можно было запустить снова. Для этого предназначен метод `reinitialize()`:

```
void reinitialize()
```

Метод `reinitialize()` сбрасывает состояние вызывающей задачи. Тем не менее, любые изменения, внесенные в постоянные данные, с которыми работала задача, не отменяются. Например, если задача модифицировала массив, то вызов `reinitialize()` произведенные ею изменения не отменяет.

Дальнейшие исследования

В предыдущем обсуждении были представлены основы инфраструктуры Fork/Join Framework и описано несколько часто используемых методов. Однако Fork/Join Framework — весьма развитая инфраструктура, которая включает в себя дополнительные возможности, обеспечивающие расширенный контроль над параллелизмом. Хотя рассмотрение всех вопросов и нюансов, связанных с параллельным программированием и Fork/Join Framework, выходит далеко за рамки книги, ниже кратко упоминаются некоторые другие возможности.

Другие избранные возможности ForkJoinTask

В ряде случаев необходимо гарантировать, что такие методы, как invokeAll() и fork(), вызываются только изнутри задачи ForkJoinTask. Обычно это несложно, но иногда приходится иметь дело с кодом, который может выполняться либо внутри, либо вне задачи. Выяснить, выполняется ли код внутри задачи, позволяет метод inForkJoinPool().

Преобразовать объект Runnable или Callable в объект ForkJoinTask можно с применением метода adapt(), определенного в ForkJoinTask, который имеет три формы, предназначенные для преобразования объекта Callable, для преобразования объекта Runnable, не возвращающего результат, и для преобразования объекта Runnable, возвращающего результат. В случае Callable запускается метод call(), а в случае Runnable — метод run().

С использованием метода getQueuedTaskCount() можно получить приблизительное количество задач, находящихся в очереди вызывающего потока. Метод getSurplusQueuedTaskCount() позволяет получить приблизительное количество задач в очереди вызывающего потока, превышающее количество других потоков в пуле, которые могут их перехватить. Вспомните, что в инфраструктуре Fork/Join Framework перехват работы представляет собой один из способов достижения высокого уровня эффективности. Хотя такой процесс является автоматическим, в некоторых случаях информация может оказаться полезной для оптимизации производительности.

В классе ForkJoinTask определены варианты методов join() и invoke(), которые начинаются с префикса quietly. Они кратко описаны в табл. 29.4.

Таблица 29.4. Варианты методов join() и invoke(), определенные в классе ForkJoinTask

Метод	Описание
final void quietlyJoin()	Присоединяется к задаче, но не возвращает результат и не генерирует исключение
final void quietlyInvoke()	Вызывает задачу, но не возвращает результат и не генерирует исключение

По существу указанные методы аналогичны своим аналогам без префикса quietly, но только они не возвращают значений и не генерируют исключений.

С помощью метода tryUnfork() можно попытаться "отменить вызов" задачи (другими словами, отменить ее планирование).

Методы вроде getForkJoinTaskTag() и setForkJoinTaskTag() поддерживают признаки — короткие целочисленные значения, связанные с задачей. Они могут быть полезны в специализированных приложениях.

Класс ForkJoinTask реализует интерфейс Serializable и потому он допускает сериализацию. Тем не менее, во время выполнения сериализация не применяется.

Другие избранные возможности ForkJoinPool

Одним из весьма полезных средств при настройке приложений Fork/Join Framework является переопределенная версия метода toString() в классе ForkJoinPool, которая отображает удобный для пользователя краткий обзор состояния пула. Чтобы увидеть его в действии, поместите показанный далее код запуска и последующего ожидания задачи в класс FJExperiment из рассмотренной ранее программы для экспериментирования с задачами:

```
// Запустить главную задачу ForkJoinTask асинхронным образом.
fjp.execute(task);

// Отобразить состояние пула во время ожидания.
while(!task.isDone()) {
  System.out.println(fjp);
}
```

Запустив программу, вы увидите на экране ряд сообщений, описывающих состояние пула. Ниже приведен пример одного из них. Конечно, ваш вывод может варьироваться в зависимости от количества процессоров, пороговых значений, рабочей нагрузки и т.д.

```
java.util.concurrent.ForkJoinPool@141d683[Running, parallelism = 2,
size = 2, active = 0, running = 2, steals = 0, tasks = 0, submissions = 1]
```

Вызвав метод isQuiescent(), можно выяснить, простаивает ли пул в данный момент. Он возвращает true, если в пуле нет активных потоков, или false в противном случае.

Метод getPoolSize() позволяет получить количество рабочих потоков, находящихся в текущий момент в пуле, а метод getActiveThreadCount() — приблизительное количество активных потоков в пуле.

Чтобы закрыть пул, необходимо вызвать метод shutdown(). Текущие активные задачи по-прежнему будут выполняться, но запускать новые задачи станет невозможно. Для немедленной остановки пула понадобится вызвать метод shutdownNow(). В этом случае предпринимается попытка отмены текущих активных задач. (Однако важно отметить, что ни один из упомянутых методов не влияет на общий пул.) Метод isShutdown() позволяет определить, закрыт ли пул. Он возвращает true, если пул был закрыт, или false в противном случае. Для выяснения, закрыт ли пул и завершены ли все задачи, нужно вызвать метод isTerminated().

Советы по использованию Fork/Join Framework

В этом разделе дается несколько советов, которые помогут избежать наиболее неприятных ловушек, связанных с использованием инфраструктуры Fork/Join Framework. Во-первых, избегайте применения слишком низкого

порога последовательной обработки. Как правило, лучше ошибиться в большую, а не в меньшую сторону. Если порог слишком низок, тогда на создание и переключение задач может уйти больше времени, чем на их обработку. Во-вторых, обычно лучше использовать стандартный уровень параллелизма. Указание меньшего числа может привести к значительному снижению выгод от применения Fork/Join Framework.

В общем случае задача `ForkJoinTask` не должна использовать синхронизированные методы или синхронизированные блоки кода. Кроме того, обычно нежелательно, чтобы в методе `calculate()` применялись другие типы синхронизации, такие как семафор. (Тем не менее, при необходимости можно использовать класс `Phaser`, поскольку он совместим с механизмом Fork/Join Framework.) Не забывайте, что основная идея `ForkJoinTask` кроется в стратегии "разделяй и властвуй", которая обычно не подходит в случаях, когда требуется внешняя синхронизация. Кроме того, избегайте ситуаций, в которых происходит значительная блокировка из-за ввода-вывода. Поэтому, как правило, задача `ForkJoinTask` не будет выполнять ввод-вывод. Проще говоря, чтобы наилучшим образом эксплуатировать Fork/Join Framework, задача должна производить вычисления, которые могут выполняться без внешней блокировки или синхронизации.

И последнее замечание: за исключением необычных обстоятельств не нужно выдвигать предположения относительно среды выполнения, в которой будет выполняться код. Другими словами, не следует предполагать, что будет доступно какое-то конкретное количество процессоров или же что на характеристики выполнения программы не повлияют другие процессы, функционирующие в то же самое время.

Сравнение утилит параллелизма и традиционного подхода к многопоточности в Java

Учитывая мощь и гибкость утилит параллелизма, вполне естественно задаться следующим вопросом: заменяют ли они традиционный подход к многопоточности и синхронизации, принятый в Java? Ответ: никоим образом! Первоначальная поддержка многопоточности и встроенные средства синхронизации по-прежнему являются механизмом, который должен применяться во многих программах на Java. Например, `synchronized`, `wait()` и `notify()` предлагают элегантные решения широкого круга задач. Однако когда требуется дополнительный контроль, то доступны утилиты параллелизма, которые с этим справятся. Кроме того, инфраструктура Fork/Join Framework предлагает эффективный способ интеграции методик параллельного программирования в более сложные приложения.

ГЛАВА

30 Потоковый API-интерфейс

На протяжении многих лет язык Java находился в процессе постоянной эволюции, и в каждом выпуске добавлялись функции, расширяющие богатство и мощь языка. Двумя особенно важными функциональными средствами считаются лямбда-выражения и потоковый API (Stream API). Лямбда-выражения были описаны в главе 15, а потоковый API рассматривается в настоящей главе. Как будет показано, потоковый API разработан с учетом лямбда-выражений. Кроме того, потоковый API обеспечивает одну из наиболее значительных демонстраций возможностей, которые лямбда-выражения привнесли в Java.

Прежде чем продолжить, необходимо сделать одно важное замечание: потоковый API использует ряд самых сложных средств Java. Для эффективной работы с ним необходимо хорошо понимать обобщения и лямбда-выражения, а также базовые концепции параллельного выполнения и Collections Framework (см. главы 14, 15, 20 и 29).

Основы потоков

Начнем с определения термина *поток* применительно к потоковому API: поток — это канал для данных. Таким образом, поток представляет последовательность объектов. Поток оперирует на источнике данных, таком как массив или коллекция. Поток сам по себе не обеспечивает хранилище для данных, а просто перемещает данные, возможно, с попутной фильтрацией, сортировкой или какой-то другой обработкой данных. Однако, как правило, сама потоковая операция не модифицирует источник данных. Например, сортировка потока не меняет порядок источника. Напротив, сортировка потока приводит к созданию нового потока, который производит отсортированный результат.

На заметку! Следует отметить, что применяемый здесь термин "поток" отличается от аналогичного термина, который употреблялся при описании классов ввода-вывода ранее в книге. Хотя концептуально поток ввода-вывода способен действовать во многом подобно одному из потоков, определенных в `java.util.stream`, они не являются одним и тем же понятием. Таким образом, термин "поток" в этой главе относится к объектам, которые основаны на одном из рассматриваемых здесь потоковых типов.

Потоковые интерфейсы

В потоковом API определено несколько потоковых интерфейсов, которые находятся в пакете `java.util.stream`, содержащемся в модуле `java.base`. В основе лежит `BaseStream`, который определяет базовую функциональность, доступную во всех потоках. `BaseStream` представляет собой обобщенный интерфейс, объявленный следующим образом:

```
interface BaseStream<T, S extends BaseStream<T, S>>
```

В `T` указывается тип элементов внутри потока, а в `S` — тип потока, расширяющий `BaseStream`. Интерфейс `BaseStream` расширяет `AutoCloseable`, что позволяет управлять потоком с помощью оператора `try` с ресурсами. Тем не менее, обычно необходимо закрывать только те потоки, источник данных которых требует закрытия (вроде потоков, подключенных к файлам). В большинстве случаев, например, когда источником данных является коллекция, закрывать поток не нужно. Методы, объявленные в `BaseStream`, кратко описаны в табл. 30.1.

Таблица 30.1. Методы, объявленные в интерфейсе `BaseStream`

Метод	Описание
`void close()`	Закрывает вызывающий поток с запуском всех зарегистрированных обработчиков событий закрытия. (Как будет показано далее, закрытия требуют лишь немногие потоки.)
`boolean isParallel()`	Возвращает `true`, если вызывающий поток параллельный, или `false`, если он последовательный
`Iterator<T> iterator()`	Получает итератор для потока и возвращает ссылку на него. (Заключительная операция.)
`S onClose (Runnable handler)`	Возвращает новый поток с обработчиком событий закрытия, указанным в `handler`, который будет вызываться при закрытии потока. (Промежуточная операция.)
`S parallel()`	Возвращает параллельный поток, основанный на вызывающем потоке. Если вызывающий поток уже параллельный, тогда он и возвращается. (Промежуточная операция.)
`S sequential()`	Возвращает последовательный поток, основанный на вызывающем потоке. Если вызывающий поток уже последовательный, тогда он и возвращается. (Промежуточная операция.)
`Spliterator<T> spliterator()`	Получает сплитератор для потока и возвращает ссылку на него. (Заключительная операция.)
`S unordered()`	Возвращает неупорядоченный поток, основанный на вызывающем потоке. Если вызывающий поток уже неупорядоченный, тогда он и возвращается. (Промежуточная операция.)

Существует несколько типов потоковых интерфейсов, производных от BaseStream. Наиболее универсальным из них является интерфейс Stream, объявленный следующим образом:

```
interface Stream<T>
```

В T указывается тип элементов в потоке. Из-за того, что Stream — обобщенный интерфейс, он используется для всех ссылочных типов. В дополнение к методам, унаследованным от BaseStream, интерфейс Stream добавляет собственные методы, часть которых кратко описана в табл. 30.2.

Таблица 30.2. Избранные методы, объявленные в интерфейсе Stream

Метод	Описание
`<R, A> R collect (Collector <? super T, A, R> collectorFunc)`	Накапливает элементы в изменяемом контейнере и возвращает его. Операция называется изменяемой редукцией. В R указывается тип результирующего контейнера, в T — тип элемента вызывающего потока, в A — внутренний накопленный тип, а в CollectorFunc — способ работы процесса накопления. (Заключительная операция.)
`long count()`	Подсчитывает количество элементов в потоке и возвращает результат. (Заключительная операция.)
`Stream<T> filter (Predicate<? super T> pred)`	Создает поток, содержащий элементы из вызывающего потока, которые удовлетворяют предикату, указанному в pred. (Промежуточная операция.)
`void forEach (Consumer<? super T> action)`	Выполняет код, указанный в action, для всех элементов в вызывающем потоке. (Заключительная операция.)
`<R> Stream<R> map (Function<? super T, ? extends R> mapFunc)`	Применяет mapFunc к элементам из вызывающего потока, выдавая новый поток, который содержит эти элементы. (Промежуточная операция.)
`DoubleStream mapToDouble( ToDoubleFunction<? super T> mapFunc)`	Применяет mapFunc к элементам из вызывающего потока, выдавая новый поток DoubleStream, который содержит эти элементы. (Промежуточная операция.)
`IntStream mapToInt (ToIntFunction<? super T> mapFunc)`	Применяет mapFunc к элементам из вызывающего потока, выдавая новый поток IntStream, который содержит эти элементы. (Промежуточная операция.)

Окончание табл. 30.2

Метод	Описание
`LongStream mapToLong (ToLongFunction<? super T> mapFunc)`	Применяет `mapFunc` к элементам из вызывающего потока, выдавая новый поток `LongStream`, который содержит эти элементы. (Промежуточная операция.)
`Optional<T> max (Comparator<? super T> comp)`	Находит и возвращает максимальный элемент в вызывающем потоке, используя указанное в `comp` упорядочение. (Заключительная операция.)
`Optional<T> min (Comparator<? super T> comp)`	Находит и возвращает минимальный элемент в вызывающем потоке, используя указанное в `comp` упорядочение. (Заключительная операция.)
`T reduce(T identityVal, BinaryOperator<T> accumulator)`	Возвращает результат, основанный на элементах в вызывающем потоке. Операция называется редукцией. (Заключительная операция.)
`Stream<T> sorted()`	Создает поток, который содержит элементы из вызывающего потока, отсортированные в естественном порядке. (Промежуточная операция.)
`Object[] toArray()`	Создает массив из элементов вызывающего потока. (Заключительная операция.)
`default List<T> toList()`	Создает неизменяемый список из элементов вызывающего потока. (Заключительная операция.)

Обратите внимание, что многие методы, описанные в табл. 30.1 и 30.2, обозначены как *заключительные* или *промежуточные* операции. Разница между ними очень важна. *Заключительная* операция *потребляет* поток и применяется для получения результата вроде нахождения минимального значения в потоке или для выполнения действия наподобие обеспечиваемого методом `forEach()`. После потребления поток нельзя использовать повторно. *Промежуточные* операции создают другой поток. Таким образом, промежуточные операции можно применять для создания *конвейера*, выполняющего последовательность действий. Еще один момент: промежуточные операции происходят не сразу. Взамен указанное действие выполняется, когда над новым потоком, созданным промежуточной операцией, выполняется заключительная операция. Такой механизм называется *ленивым поведением*, а промежуточные операции — *ленивыми*. Ленивое поведение позволяет потоковому API работать более эффективно.

Еще один ключевой аспект потоков связан с тем, что одни промежуточные операции не запоминают состояние, а другие его сохраняют. В операции, не запоминающей состояние, каждый элемент обрабатывается независимо от других. В операции с сохранением состояния обработка элемента может зависеть от аспектов других элементов. Скажем, сортировка представляет собой операцию с запоминанием состояния, т.к. порядок элементов зависит от значений других элементов. Таким образом, метод `sorted()` запоминает состояние. Однако элементы фильтрации, основанные на предикате без сохранения состояния, не запоминают состояние, потому что каждый элемент обрабатывается отдельно. В итоге метод `filter()` может (и должен) быть без состояния. Разница между операциями без сохранения состояния и операциями с отслеживанием состояния особенно важна, когда требуется параллельная обработка потока, поскольку для выполнения операции с отслеживанием состояния может потребоваться более одного прохода.

Поскольку интерфейс `Stream` работает с объектными ссылками, он не может напрямую оперировать с примитивными типами. Для обработки потоков примитивных типов в потоковом API определены следующие интерфейсы:

```
DoubleStream
IntStream
LongStream
```

Все перечисленные интерфейсы расширяют `BaseStream` и обладают такими же возможностями, как `Stream`, но имеют дело с примитивными, а не ссылочными типами. Вдобавок они предоставляют ряд удобных методов вроде `boxed()`, облегчающих их применение. С учетом того, что потоки объектов более распространены, основное внимание в главе уделяется интерфейсу `Stream`, но потоки примитивных типов могут использоваться в основном аналогично.

Получение потока

Получить поток можно несколькими способами. Пожалуй, чаще всего поток получается из коллекции. Начиная с версии JDK 8, интерфейс `Collection` был расширен за счет включения двух методов, которые получают поток из коллекции. Первый метод — `stream()`:

```
default Stream<E> stream()
```

Его стандартная реализация возвращает последовательный поток. Второй метод — `parallelStream()`:

```
default Stream<E> parallelStream()
```

Его стандартная реализация возвращает параллельный поток, когда это возможно. (Если параллельный поток получить невозможно, то вместо него может быть возвращен последовательный поток.) Параллельные потоки поддерживают параллельное выполнение потоковых операций. Поскольку интерфейс `Collection` реализуется каждой коллекцией, упомянутые методы можно применять для получения потока из любого класса коллекции, такого как `ArrayList` или `HashSet`.

Поток также можно получить из массива с помощью статического метода stream(), добавленного в класс Arrays. Вот одна из его форм:

```
static <T> Stream<T> stream(T[] array)
```

Метод stream() возвращает последовательный поток для элементов в массиве array. Например, следующий код получает поток для массива addresses типа Address:

```
Stream<Address> addrStrm = Arrays.stream(addresses);
```

Для метода stream() определено несколько перегруженных версий, которые обрабатывают, например, массивы примитивных типов. Они возвращают поток IntStream, DoubleStream или LongStream.

Потоки можно получать множеством других способов. Например, многие потоковые операции возвращают новый поток, а поток для источника ввода-вывода легко получить, вызвав метод lines() на экземпляре BufferedReader. Независимо от того, как был получен поток, с ним можно работать таким же способом, как с любым другим потоком.

Простой пример использования потока

Прежде чем двигаться дальше, давайте рассмотрим пример применения потока. В показанной далее программе создается список ArrayList по имени myList, содержащий набор целых чисел (которые автоматически упаковываются в ссылочный тип Integer). Затем получается поток, использующий myList в качестве источника, и демонстрируются разнообразные потоковые операции.

```
// Демонстрация потоковых операций.

import java.util.*;
import java.util.stream.*;

class StreamDemo {

  public static void main(String[] args) {

    // Создать список значений типа Integer.
    ArrayList<Integer> myList = new ArrayList<>();
    myList.add(7);
    myList.add(18);
    myList.add(10);
    myList.add(24);
    myList.add(17);
    myList.add(5);

    System.out.println("Исходный список: " + myList);

    // Получить поток для ArrayList.
    Stream<Integer> myStream = myList.stream();

    // Получить минимальное и максимальное значения с использованием
    // методов min(), max(), isPresent() и get().
    Optional<Integer> minVal = myStream.min(Integer::compare);
    if(minVal.isPresent()) System.out.println("Минимальное значение: " +
                                              minVal.get());
```

```
    // Требуется получить новый поток, поскольку предыдущий вызов min()
    // является заключительной операцией, которая потребляет поток.
    myStream = myList.stream();
    Optional<Integer> maxVal = myStream.max(Integer::compare);
    if(maxVal.isPresent()) System.out.println("Максимальное значение: " +
                                              maxVal.get());

    // Отсортировать поток с применением sorted().
    Stream<Integer> sortedStream = myList.stream().sorted();

    // Отобразить отсортированный поток с использованием forEach().
    System.out.print("Отсортированный поток: ");
    sortedStream.forEach((n) -> System.out.print(n + " "));
    System.out.println();

    // Отобразить только нечетные значения с применением filter().
    Stream<Integer> oddVals =
      myList.stream().sorted().filter((n) -> (n % 2) == 1);
    System.out.print("Нечетные значения: ");
    oddVals.forEach((n) -> System.out.print(n + " "));
    System.out.println();

    // Отобразить только нечетные значения, которые больше 5.
    // Обратите внимание,
    // что две операции фильтрования соединены в конвейер.
    oddVals = myList.stream().filter((n) -> (n % 2) == 1)
                             .filter((n) -> n > 5);
    System.out.print("Нечетные значения, которые больше 5: ");
    oddVals.forEach((n) -> System.out.print(n + " "));
    System.out.println();
  }
}
```

Ниже представлен вывод, генерируемый программой:

```
Исходный список: [7, 18, 10, 24, 17, 5]
Минимальное значение: 5
Максимальное значение: 24
Отсортированный поток: 5 7 10 17 18 24
Нечетные значения: 5 7 17
Нечетные значения, которые больше 5: 7 17
```

Давайте внимательно взглянем на каждую потоковую операцию. После создания `ArrayList` программа получает поток для списка, вызывая метод `stream()`:

```
Stream<Integer> myStream = myList.stream();
```

Как объяснялось ранее, в интерфейсе `Collection` определен метод `stream()`, который получает поток из вызывающей коллекции. Так как интерфейс `Collection` реализуется каждым классом коллекции, с помощью метода `stream()` можно получать поток для любого типа коллекции, включая применяемый здесь тип `ArrayList`. В данном случае ссылка на поток присваивается переменной `myStream`.

Далее получается и отображается минимальное значение в потоке (которое вполне естественно является минимальным значением в источнике данных):

```
Optional<Integer> minVal = myStream.min(Integer::compare);
if(minVal.isPresent()) System.out.println("Минимальное значение: " +
                                          minVal.get());
```

Вспомните из табл. 30.2, что метод `min()` объявлен следующим образом:

```
Optional<T> min(Comparator<? super T> comp)
```

Первым делом обратите внимание на тип параметра в методе `min()` — `Comparator`. Данный компаратор используется для сравнения двух элементов в потоке. В примере методу `min()` передается ссылка на метод `compare()` класса `Integer`, который применяется для реализации экземпляра `Comparator`, способного сравнивать два целых числа. Вдобавок обратите внимание на возвращаемый тип метода `min()` — `Optional`. Класс `Optional` был описан в главе 21, но стоит напомнить, как он работает. Обобщенный класс `Optional` находится в пакете `java.util` и имеет такое объявление:

```
class Optional<T>
```

В `T` указывается тип элемента. Экземпляр `Optional` может либо содержать значение типа `T`, либо быть пустым. Для выяснения, присутствует ли значение, можно использовать метод `isPresent()`. Если значение доступно, тогда его можно получить с помощью метода `get()` или метода `orElseThrow()` в случае работы с JDK 10 и более поздней версией. В примере применяется `get()`. Возвращаемый объект будет содержать минимальное значение потока в виде объекта `Integer`.

Еще одно замечание: `min()` — заключительная операция, потребляющая поток. Таким образом, после выполнения `min()` поток `myStream` нельзя использовать повторно.

Следующий код получает и отображает максимальное значение в потоке:

```
myStream = myList.stream();
Optional<Integer> maxVal = myStream.max(Integer::compare);
if(maxVal.isPresent()) System.out.println("Максимальное значение: " +
                                          maxVal.get());
```

Переменной `myStream` снова присваивается поток, возвращаемый вызовом `myList.stream()`. Как только что упоминалось, поступать так необходимо из-за того, что предыдущий вызов `min()` потребляет предыдущий поток и потому требуется новый поток. Затем вызывается метод `max()` для получения максимального значения. Подобно `min()` метод `max()` возвращает объект `Optional`, значение которого получается вызовом `get()`.

Далее в программе получается отсортированный поток:

```
Stream<Integer> sortedStream = myList.stream().sorted();
```

Здесь метод `sorted()` вызывается на потоке, который возвращает `myList.stream()`. Поскольку вызов `sorted()` — промежуточная операция, ее результатом будет новый поток, который присваивается `sortedStream`. Содержимое отсортированного потока отображается с помощью `forEach()`:

```
sortedStream.forEach((n) -> System.out.print(n + " "));
```

Метод forEach() выполняет операцию в отношении каждого элемента внутри потока. В данном случае он просто вызывает System.out.print() для каждого элемента в sortedStream. Цель достигается с помощью лямбда-выражения. Метод forEach() имеет следующую общую форму:

```
void forEach(Consumer<? super T> action)
```

Обобщенный функциональный интерфейс Consumer находится в пакете java.util.function. Вот его абстрактный метод accept():

```
void accept(T objRef)
```

Лямбда-выражение в вызове forEach() предоставляет реализацию accept(). Метод forEach() является заключительной операцией, т.е. после его завершения поток потребляется.

Затем отсортированный поток фильтруется посредством filter(), так что он содержит только нечетные значения:

```
Stream<Integer> oddVals =
  myList.stream().sorted().filter((n) -> (n % 2) == 1);
```

Метод filter() фильтрует поток на основе предиката и возвращает новый поток, содержащий только те элементы, которые удовлетворяют предикату:

```
Stream<T> filter(Predicate<? super T> pred)
```

Обобщенный функциональный интерфейс Predicate находится в пакете java.util.function. Ниже показан его абстрактный метод test():

```
boolean test(T objRef)
```

Он возвращает true, если объект, на который ссылается objRef, удовлетворяет предикату, или false в противном случае. Данный метод реализуется лямбда-выражением, передаваемым в filter(). Так как метод filter() — промежуточная операция, он возвращает новый поток, содержащий отфильтрованные значения, которые в рассматриваемом примере представляют собой нечетные числа. Далее элементы отображаются с помощью forEach(), как и ранее.

Поскольку filter() или любая другая промежуточная операция возвращает новый поток, отфильтрованный поток можно снова отфильтровать, как демонстрируется в следующем фрагменте кода, где создается поток, содержащий только нечетные значения, которые больше 5:

```
oddVals = myList.stream().filter((n) -> (n % 2) == 1)
                         .filter((n) -> n > 5);
```

Обратите внимание, что обоим фильтрам передаются лямбда-выражения.

Операции редукции

Рассмотрим методы min() и max() из предыдущего примера программы. Оба они являются заключительными операциями, которые возвращают результат на основе элементов в потоке. В контексте потокового API они представляют *операции редукции*, т.к. каждая из них сводит поток к единственному

значению — к минимуму и максимуму в данном случае. В потоковом API они называются операциями редукции *частного случая*, потому что выполняют специфическую функцию. Помимо `min()` и `max()` также доступны другие операции редукции частного случая, такие как метод `count()`, который подсчитывает количество элементов в потоке. Тем не менее, потоковый API обобщает эту концепцию, предоставляя метод `reduce()`. С применением `reduce()` можно возвратить значение из потока на основе произвольных критериев. По определению все операции редукции являются заключительными.

В интерфейсе `Stream` определены три формы метода `reduce()`. Сначала будут использоваться две из них:

```
Optional<T> reduce(BinaryOperator<T> accumulator)
T reduce(T identityVal, BinaryOperator<T> accumulator)
```

Первая форма возвращает объект типа `Optional`, который содержит результат. Вторая форма возвращает объект типа `T` (т.е. тип элементов потока). В обеих формах `accumulator` представляет собой аккумулирующую функцию, которая работает с двумя значениями и выдает результат. Во второй форме `identityVal` — значение идентичности, такое что аккумулирующая операция над `identityVal` и любым элементом потока выдает этот элемент без изменений. Например, в случае операции сложения значением `identityVal` будет 0, т.к. 0 + x равно x, а для операции умножения значением `identityVal` будет 1, потому что 1 * x равно x.

Функциональный интерфейс `BinaryOperator`, объявленный в пакете `java.util.function`, расширяет функциональный интерфейс `BiFunction`. В `BiFunction` определен следующий абстрактный метод:

```
R apply(T val, U val2)
```

В `R` указывается тип результата, в `T` — тип первого операнда, а в `U` — тип второго операнда. Таким образом, `apply()` применяет функцию к двум своим операндам (`val` и `val2`) и возвращает результат. Когда интерфейс `BinaryOperator` расширяет `BiFunction`, он указывает один и тот же тип для всех параметров типа. Таким образом, вот как выглядит метод `apply()` применительно к `BinaryOperator`:

```
T apply(T val, T val2)
```

Что касается метода `reduce()`, то `val` будет содержать предыдущий результат, а `val2` — следующий элемент. В зависимости от используемой версии `reduce()` при первом вызове `val` получит либо значение `identityVal`, либо значение первого элемента.

Важно понимать, что аккумулирующая операция должна удовлетворять трем ограничениям:

- не запоминать состояние;
- не создавать помехи;
- быть ассоциативной.

Как объяснялось ранее, *отсутствие запоминания состояния* означает, что операция не полагается на какую-либо информацию о состоянии. Таким образом, каждый элемент обрабатывается независимо. *Отсутствие помех* означает, что операция не изменяет источник данных. Наконец, операция обязана быть *ассоциативной*. Термин "ассоциативная" применяется в своем обычном арифметическом смысле, т.е. для заданной ассоциативной операции, используемой в последовательности операций, не имеет значения, какая пара операндов обрабатывается первой. Например, конструкция

```
(10 * 2) * 7
```

даст такой же результат, как и конструкция

```
10 * (2 * 7)
```

Ассоциативность особенно важна в случае применения операций редукции в параллельных потоках, обсуждаемых в следующем разделе.

Ниже приведен пример программы, демонстрирующей использование только что описанных версий `reduce()`:

```
// Демонстрация использования метода reduce().
import java.util.*;
import java.util.stream.*;

class StreamDemo2 {
  public static void main(String[] args) {
    // Создать список значений типа Integer.
    ArrayList<Integer> myList = new ArrayList<>();
    myList.add(7);
    myList.add(18);
    myList.add(10);
    myList.add(24);
    myList.add(17);
    myList.add(5);

    // Два способа получения целочисленного произведения элементов
    // в myList с использованием reduce().
    Optional<Integer> productObj = myList.stream().reduce((a,b) -> a*b);
    if(productObj.isPresent())
      System.out.println("Произведение как Optional: " + productObj.get());

    int product = myList.stream().reduce(1, (a,b) -> a*b);
    System.out.println("Произведение как int: " + product);
  }
}
```

В выводе видно, что в обоих случаях применения метода `reduce()` получается один и тот же результат:

```
Произведение как Optional: 2570400
Произведение как int: 2570400
```

В первой версии метода `reduce()` для получения произведения двух значений используется лямбда-выражение. Поскольку поток в данном случае содержит значения `Integer`, объекты `Integer` автоматически распаковыва-

ются для умножения и заново упаковываются для возвращения результата. Два значения представляют текущий результат и следующий элемент в потоке. Окончательный результат возвращается в виде объекта типа `Optional`. Значение получается вызовом `get()` на возвращенном объекте.

Во второй версии явно указано значение идентичности, которое равно 1 для умножения. Обратите внимание, что результат возвращается в виде объекта типа элемента — `Integer` в данном случае.

Хотя простые операции редукции вроде умножения полезны в качестве примеров, редукция этим не ограничивается. Например, продолжая предыдущую программу, следующий код получает произведение только четных значений:

```
int evenProduct = myList.stream().reduce(1, (a,b) -> {
                    if(b%2 == 0) return a*b; else return a;
                  });
```

Обратите особое внимание на лямбда-выражение. Если `b` имеет четное значение, то возвращается `a*b`. В противном случае возвращается `a`. Прием работает, потому что `a` содержит текущий результат, а `b` — следующий элемент, как объяснялось ранее.

Использование параллельных потоков

Перед дальнейшим исследованием потокового API полезно обсудить параллельные потоки. Как было указано ранее в книге, параллельное выполнение кода на многоядерных процессорах может привести к значительному повышению производительности. По этой причине параллельное программирование стало важной частью работы современного программиста. Однако параллельное программирование может оказаться сложным и подверженным ошибкам. Одно из преимуществ потокового API связано с возможностью простой и надежной параллельной обработки определенных операций.

Параллельную обработку потока запросить довольно просто: достаточно воспользоваться параллельным потоком. Как упоминалось ранее, один из способов получения параллельного потока предусматривает применение метода `parallelStream()`, определенного в `Collection`, а другой способ — вызов метода `parallel()` на последовательном потоке. Метод `parallel()` определен в `BaseStream`:

```
S parallel()
```

Метод `parallel()` возвращает параллельный поток, основанный на последовательном потоке, для которого он был вызван. (Если он вызывается на потоке, который уже является параллельным, тогда возвращается вызывающий поток.) Разумеется, важно понимать, что даже при параллельном потоке параллелизм будет обеспечен только в случае его поддержки средой.

После получения параллельного потока потоковые операции могут выполняться параллельно, при условии, что параллелизм поддерживается средой.

Например, первую операцию reduce() в предыдущей программе можно распараллелить, заменив parallelStream() вызовом stream():

```
Optional<Integer> productObj = myList.parallelStream().reduce((a,b) -> a*b);
```

Результаты будут теми же, но умножения могут происходить в разных потоках. Как правило, любая операция, используемая в отношении параллельного потока, не должна запоминать состояние, а также не создавать помехи и быть ассоциативной. Это гарантирует, что результаты, полученные при выполнении операций в параллельном потоке, будут такими же, как результаты, полученные при выполнении тех же операций в последовательном потоке.

В случае применения параллельных потоков следующая версия метода reduce() может оказаться особенно полезной. Она предоставляет способ указать способ объединения частичных результатов:

```
<U> U reduce(U identityVal, BiFunction<U, ? super T, U> accumulator
  BinaryOperator<U> combiner)
```

Здесь в combiner определена функция, которая объединяет два значения, созданные функцией accumulator. Продолжая предыдущую программу, следующий оператор вычисляет произведение элементов в myList с использованием параллельного потока:

```
int parallelProduct = myList.parallelStream().reduce(1, (a,b) -> a*b,
                                                     (a,b) -> a*b);
```

В примере видно, что аккумулирующая и объединяющая функции делают одно и то же. Тем не менее, бывают случаи, когда действия аккумулирующей функции должны отличаться от действий объединяющей функции. Рассмотрим приведенную ниже программу. В ней myList содержит список значений типа double. Затем с применением объединяющей версии reduce() вычисляется произведение квадратных корней каждого элемента в списке.

```
// Демонстрация использования объединяющей функции посредством reduce().
import java.util.*;
import java.util.stream.*;
class StreamDemo3 {
  public static void main(String[] args) {
    // Список значений double.
    ArrayList<Double> myList = new ArrayList<>();
    myList.add(7.0);
    myList.add(18.0);
    myList.add(10.0);
    myList.add(24.0);
    myList.add(17.0);
    myList.add(5.0);
    double productOfSqrRoots = myList.parallelStream().reduce(
                                 1.0,
                                 (a,b) -> a * Math.sqrt(b),
                                 (a,b) -> a * b
                               );
    System.out.println("Произведение квадратных корней: " + productOfSqrRoots);
  }
}
```

Обратите внимание, что аккумулирующая функция умножает квадратные корни двух элементов, а объединяющая функция перемножает частичные результаты. Следовательно, эти две функции различаются. Более того, чтобы вычисление работало корректно, они *обязаны* различаться. Например, при попытке получить произведение квадратных корней элементов с помощью следующего оператора, возникнет ошибка:

```
// Не работает.
double productOfSqrRoots2 = myList.parallelStream().reduce(
                            1.0,
                            (a,b) -> a * Math.sqrt(b));
```

В данной версии `reduce()` аккумулирующая и объединяющая функции совпадают, что приводит к ошибке, т.к. при объединении двух частичных результатов перемножаются их квадратные корни, а не сами частичные результаты. Интересно отметить, что если изменить поток в предыдущем вызове `reduce()` на последовательный, то операция дала бы правильный ответ, поскольку отсутствовала бы потребность в объединении двух частичных результатов. Проблема возникает при использовании параллельного потока.

Переключить параллельный поток на последовательный можно с помощью метода `sequence()`, определенного в `BaseStream`:

```
S sequential()
```

Как правило, поток по мере необходимости можно переключать между параллельной и последовательной версией.

Есть еще один аспект, касающийся потока, о котором следует помнить во время применения параллельного выполнения — порядок элементов. Потоки могут быть либо упорядоченными, либо неупорядоченными. В общем случае, если источник данных упорядочен, то и поток будет упорядоченным. Однако при использовании параллельного потока иногда можно увеличить производительность, позволив потоку быть неупорядоченным. Когда параллельный поток неупорядочен, каждый раздел потока может работать независимо, не координируя свои действия с другими. В случаях, когда порядок операций не имеет значения, можно установить неупорядоченное поведение, вызвав метод `unordered()`:

```
S unordered()
```

И еще один момент: метод `forEach()` может не предохранять порядок параллельного потока. При желании выполнить операцию над каждым элементом в параллельном потоке с предохранением порядка имеет смысл подумать о применении метода `forEachOrdered()`, который используется аналогично методу `forEach()`.

Сопоставление

Часто удобно сопоставлять элементы одного потока с другим. Скажем, для потока с базой данных, содержащей имена, телефонные номера и адреса электронной почты, можно было бы организовать сопоставление имен и

адресов электронной почты с другим потоком. Еще одним примером может служить применение к элементам в потоке некоторого преобразования, для чего преобразованные элементы понадобится сопоставить с элементами в новом потоке. Поскольку операции сопоставления довольно распространены, потоковый API обеспечивает для них встроенную поддержку. Наиболее универсальным методом сопоставления является `map()`:

```
<R> Stream<R> map(Function<? super T, ? extends R> mapFunc)
```

В `R` указывается тип элементов нового потока, в `T` — тип элементов вызывающего потока, а в `mapFunc` — экземпляр `Function`, который выполняет сопоставление. Функция сопоставления должна не запоминать состояние и не создавать помехи. Так как возвращается новый поток, `map()` является промежуточным методом.

Функциональный интерфейс `Function` находится в пакете `java.util.function` и объявлен следующим образом:

```
Function<T, R>
```

Что касается `map()`, то в `T` указывается тип элемента, а в `R` — результат сопоставления. В интерфейсе `Function` есть такой абстрактный метод:

```
R apply(T val)
```

В `val` передается ссылка на сопоставляемый объект. Возвращается сопоставленный результат.

Ниже приведен простой пример использования метода `map()` в виде варианта предыдущей программы. Как и прежде, программа вычисляет произведение квадратных корней значений в `ArrayList`. Однако в данной версии квадратные корни элементов сначала сопоставляются с новым потоком, после чего для вычисления произведения применяется метод `reduce()`.

```
// Сопоставление одного потока с другим.
import java.util.*;
import java.util.stream.*;
class StreamDemo4 {
  public static void main(String[] args) {
    // Список значений типа double.
    ArrayList<Double> myList = new ArrayList<>();
    myList.add(7.0);
    myList.add(18.0);
    myList.add(10.0);
    myList.add(24.0);
    myList.add(17.0);
    myList.add(5.0);

    // Сопоставить квадратные корни элементов в myList с новым потоком.
    Stream<Double> sqrtRootStrm = myList.stream().map((a) -> Math.sqrt(a));

    // Найти произведение квадратных корней.
    double productOfSqrRoots = sqrtRootStrm.reduce(1.0, (a,b) -> a*b);

    System.out.println("Произведение квадратных корней: " + productOfSqrRoots);
  }
}
```

Вывод программы остается таким же, как и ранее. Разница между новой версией и предыдущей заключается просто в том, что преобразование (т.е. вычисление квадратных корней) происходит во время сопоставления, а не редукции. В итоге для вычисления произведения можно использовать форму `reduce()` с двумя параметрами, т.к. больше нет необходимости предоставлять отдельную объединяющую функцию.

Ниже приведен пример применения метода `map()` для создания нового потока с избранными полями исходного потока. В данном случае исходный поток содержит объекты типа `NamePhoneEmail` с именами, телефонными номерами и адресами электронной почты. Затем выполняется сопоставление имен и телефонных номеров с новым потоком объектов `NamePhone`, а адреса электронной почты отбрасываются.

```
// Использование метода map() для создания нового потока,
// который содержит избранные значения исходного потока.
import java.util.*;
import java.util.stream.*;

class NamePhoneEmail {
  String name;
  String phonenum;
  String email;

  NamePhoneEmail(String n, String p, String e) {
    name = n;
    phonenum = p;
    email = e;
  }
}

class NamePhone {
  String name;
  String phonenum;

  NamePhone(String n, String p) {
    name = n;
    phonenum = p;
  }
}

class StreamDemo5 {
  public static void main(String[] args) {
    // Список имен, телефонных номеров и адресов электронной почты.
    ArrayList<NamePhoneEmail> myList = new ArrayList<>();

    myList.add(new NamePhoneEmail("Larry", "555-5555",
                                  "Larry@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("James", "555-4444",
                                  "James@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("Mary", "555-3333",
                                  "Mary@HerbSchildt.com"));

    System.out.println("Исходные значения в myList: ");
    myList.stream().forEach((a) -> {
      System.out.println(a.name + " " + a.phonenum + " " + a.email);
    });
```

```
    System.out.println();

    // Сопоставить с новым потоком только имена и телефонные номера.
    Stream<NamePhone> nameAndPhone = myList.stream().map(
                                  (a) -> new NamePhone(a.name,a.phonenum)
                                      );

    System.out.println("Список имен и телефонных номеров: ");
    nameAndPhone.forEach( (a) -> {
      System.out.println(a.name + " " + a.phonenum);
    });
  }
}
```

Вывод, генерируемый программой, подтверждает сопоставление:

```
Исходные значения в myList:
Larry 555-5555 Larry@HerbSchildt.com
James 555-4444 James@HerbSchildt.com
Mary 555-3333 Mary@HerbSchildt.com

Список имен и телефонных номеров:
Larry 555-5555
James 555-4444
Mary 555-3333
```

Поскольку несколько промежуточных операций можно объединить в конвейер, легко создавать крайне эффективные действия. Например, в следующем операторе используются методы `filter()` и `map()` для создания нового потока, который содержит только имя и телефонный номер, относящиеся к элементам `"James"`:

```
Stream<NamePhone> nameAndPhone = myList.stream().
                                 filter((a) -> a.name.equals("James")).
                                 map((a) -> new NamePhone(a.name,a.phonenum));
```

Такой вид фильтрации очень распространен при создании запросов в стиле базы данных. По мере приобретения опыта работы с потоковым API вы обнаружите, что цепочки операций подобного рода можно применять для создания сложных запросов, слияний и выборок в потоке данных.

Помимо только что описанной версии доступны еще три формы метода `map()`, которые возвращают поток примитивного типа:

```
IntStream mapToInt(ToIntFunction<? super T> mapFunc)
LongStream mapToLong(ToLongFunction<? super T> mapFunc)
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapFunc)
```

В `mapFunc` должен быть реализован абстрактный метод, который определен заданным интерфейсом и возвращает значение соответствующего типа. Например, `ToDoubleFunction` указывает метод `applyAsDouble(T val)`, который обязан возвращать значение своего параметра как значение типа `double`.

Ниже показан пример использования потока примитивного типа, где сначала создается список `ArrayList` значений `Double`, после чего с применением методов `stream()` и `mapToInt()` создается поток `IntStream`, содержащий ближайшие бо́льшие целые для каждого значения.

```
// Сопоставление Stream с IntStream.

import java.util.*;
import java.util.stream.*;

class StreamDemo6 {
  public static void main(String[] args) {

    // Список значений типа double.
    ArrayList<Double> myList = new ArrayList<>();
    myList.add(1.1);
    myList.add(3.6);
    myList.add(9.2);
    myList.add(4.7);
    myList.add(12.1);
    myList.add(5.0);

    System.out.print("Исходные элементы myList: ");
    myList.stream().forEach( (a) -> {
      System.out.print(a + " ");
    });
    System.out.println();

    // Сопоставить ближайшие большие целые значения для элементов
    // в myList с IntStream.
    IntStream cStrm = myList.stream().mapToInt((a) -> (int) Math.ceil(a));

    System.out.print("Ближайшие большие целые значения для элементов myList: ");
    cStrm.forEach( (a) -> {
      System.out.print(a + " ");
    });
  }
}
```

Вот вывод, генерируемый программой:

```
Исходные значения myList: 1.1 3.6 9.2 4.7 12.1 5.0
Ближайшие большие целые значения для элементов myList: 2 4 10 5 13 5
```

Поток, созданный методом `mapToInt()`, содержит ближайшие большие целые значения для исходных элементов в `myList`.

Прежде чем закончить обсуждение сопоставления, необходимо отметить, что потоковый API также предоставляет методы, поддерживающие *плоские карты*: `flatMap()`, `flatMapToInt()`, `flatMapToLong()` и `flatMapToDouble()`. Методы плоских карт рассчитаны на ситуации, когда каждый элемент в исходном потоке отображается на несколько элементов в результирующем потоке. Начиная с версии JDK 16, класс `Stream` предоставляет дополнительные методы, связанные с плоскими картами: `mapMulti()`, `mapMultiToInt()`, `mapMultiToLong()` и `mapMulltiToDouble()`.

Накопление

Как демонстрировалось в предшествующих примерах, можно получать поток из коллекции (что на самом деле делается часто), но временами желательно обратное: получить коллекцию из потока. Для выполнения такого действия обычно используется метод `collect()`, который имеет две формы.

Первой будет применяться следующая форма:

```
<R, A> R collect(Collector<? super T, A, R> collectorFunc)
```

В R указывается тип результата, в T — тип элемента вызывающего потока, в A — внутренний накопленный тип, а в CollectorFunc — способ работы процесса накопления. Метод collect() является заключительной операцией.

Интерфейс Collector объявлен в пакете java.util.stream, как показано ниже:

```
interface Collector<T, A, R>
```

Типы T, A и R были описаны ранее. В интерфейсе Collector определено несколько методов, но здесь реализовывать их не нужно. Взамен будут использоваться два предопределенных накопителя, предоставляемые классом Collectors из пакета java.util.stream.

В классе Collectors определены статические методы накопления, которые можно применять в том виде как есть. Далее будут использоваться два метода — toList() и toSet():

```
static <T> Collector<T, ?, List<T>> toList()
static <T> Collector<T, ?, Set<T>> toSet()
```

Метод toList() возвращает накопитель, который можно применять для сбора элементов в объект List. Метод toSet() возвращает накопитель, который можно использовать для сбора элементов в объект Set. Например, вот как вызвать метод collect(), чтобы накопить элементы в объекте List:

```
collect(Collectors.toList())
```

В следующей программе переделан пример из предыдущего раздела для накопления имен и телефонных номеров в объектах List и Set:

```
// Использование collect() для создания объектов List и Set из списка.
import java.util.*;
import java.util.stream.*;

class NamePhoneEmail {
  String name;
  String phonenum;
  String email;

  NamePhoneEmail(String n, String p, String e) {
    name = n;
    phonenum = p;
    email = e;
  }
}

class NamePhone {
  String name;
  String phonenum;

  NamePhone(String n, String p) {
    name = n;
    phonenum = p;
  }
}
```

```
class StreamDemo7 {
  public static void main(String[] args) {
    // Список имен, телефонных номеров и адресов электронной почты.
    ArrayList<NamePhoneEmail> myList = new ArrayList<>();
    myList.add(new NamePhoneEmail("Larry", "555-5555",
                                  "Larry@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("James", "555-4444",
                                  "James@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("Mary", "555-3333",
                                  "Mary@HerbSchildt.com"));
    // Сопоставить с новым потоком имена и телефонные номера.
    Stream<NamePhone> nameAndPhone = myList.stream().map(
                                (a) -> new NamePhone(a.name,a.phonenum)
                                   );
    // Использовать collect() для создания списка List
    // с именами и телефонными номерами.
    List<NamePhone> npList = nameAndPhone.collect(Collectors.toList());
    System.out.println("Имена и телефонные номера в List:");
    for(NamePhone e : npList)
      System.out.println(e.name + ": " + e.phonenum);
    // Получить еще одно сопоставление имен и телефонных номеров.
    nameAndPhone = myList.stream().map(
                              (a) -> new NamePhone(a.name,a.phonenum)
                                    );
    // Создать набор Set с применением collect().
    Set<NamePhone> npSet = nameAndPhone.collect(Collectors.toSet());
    System.out.println("\nИмена и телефонные номера в Set:");
    for(NamePhone e : npSet)
      System.out.println(e.name + ": " + e.phonenum);
  }
}
```

Ниже приведен вывод из программы:

```
Имена и телефонные номера в List:
Larry: 555-5555
James: 555-4444
Mary: 555-3333

Имена и телефонные номера в Set:
James: 555-4444
Larry: 555-5555
Mary: 555-3333
```

С помощью следующей строки в программе производится накопление имен и телефонных номеров в объект `List` посредством `toList()`:

```
List<NamePhone> npList = nameAndPhone.collect(Collectors.toList());
```

После выполнения этой строки коллекцию, на которую ссылается `npList`, можно применять как любую другую коллекцию типа `List`. Скажем, по ней можно проходить в цикле `for` стиля "for-each":

```
for(NamePhone e : npList)
  System.out.println(e.name + ": " + e.phonenum);
```

Объект Set создается аналогично через collect(Collectors.toSet()). Возможность перемещения данных из коллекции в поток и затем обратно в коллекцию — очень мощная характеристика потокового API, которая дает возможность работать с коллекцией через поток, а затем переупаковывать ее как коллекцию. Более того, при необходимости потоковые операции могут выполняться параллельно.

Версия метода collect(), используемая в предыдущем примере, довольно удобна и часто она полностью подходит, но есть и вторая версия, которая обеспечивает больший контроль над процессом накопления:

```
<R> R collect(Supplier<R> target, BiConsumer<R, ? super T> accumulator,
              BiConsumer <R, R> combiner)
```

В target задается способ создания объекта, содержащего результат. Скажем, для применения класса LinkedList в качестве результирующей коллекции необходимо указать его конструктор. Функция accumulator добавляет элемент к результату, а combiner объединяет два частичных результата. Таким образом, функции accumulator и combiner работают так же, как и в методе reduce(). Они обе не должны запоминать состояние и не создавать помехи, а также быть ассоциативными.

Обратите внимание, что параметр target имеет тип Supplier — функциональный интерфейс, объявленный в java.util.function. В нем определен только метод get(), который не имеет параметров и в данном случае возвращает объект типа R. Таким образом, что касается collect(), то метод get() возвращает ссылку на изменяемый объект хранения, такой как коллекция.

Также важно отметить, что типом accumulator и combiner является BiConsumer — функциональный интерфейс из пакета java.util.function, в котором определен абстрактный метод accept():

```
void accept(T obj, U obj2)
```

Метод accept() выполняет операцию над obj и obj2. Для accumulator в obj указывается целевая коллекция, а в obj2 — элемент, который нужно добавить в эту коллекцию. Для combiner в obj и obj2 задаются две коллекции, которые будут объединены.

Только что описанная версия collect() позволяет указать в качестве цели в предыдущей программе объект LinkedList:

```
LinkedList<NamePhone> npList = nameAndPhone.collect(
                               () -> new LinkedList<>(),
                               (list, element) -> list.add(element),
                               (listA,listB ) -> listA.addAll(listB));
```

Обратите внимание, что в первом аргументе collect() передается лямбда-выражение, которое возвращает новый объект LinkedList. Во втором аргументе используется стандартный метод add() для добавления элемента в список. В третьем элементе применяется метод addAll() для объединения

двух связных списков. Интересно отметить возможность использования любого метода, определенного в LinkedList, для добавления элемента в список. Например, можно применить addFirst(), чтобы добавить элементы в начало списка:

```
(list, element) -> list.addFirst(element)
```

Как вы уже догадались, указывать лямбда-выражение для аргументов метода collect() нужно не всегда. Часто вполне достаточно ссылок на методы и/или конструкторы. Например, следующий код создает объект HashSet, который содержит все элементы в потоке nameAndPhone:

```
HashSet<NamePhone> npSet = nameAndPhone.collect(HashSet::new,
                                                HashSet::add,
                                                HashSet::addAll);
```

Обратите внимание, что в первом аргументе указана ссылка на конструктор HashSet, а во втором и третьем — ссылки на методы add() и addAll() класса HashSet.

И последнее замечание: в контексте потокового API метод collect() выполняет то, что называется *изменяемой редукцией*. Дело в том, что результатом редукции является изменяемый объект хранения, такой как коллекция. Когда необходимо получить неизменяемую коллекцию из потока, начиная с версии JDK 16, можно использовать метод toList() класса Stream, который возвращает неизменяемый список.

Итераторы и потоки

Хотя поток не является объектом хранения данных, все же с помощью итератора можно проходить по его элементам почти так же, как в случае применения итератора для прохода по элементам коллекции. В потоковом API поддерживаются два типа итераторов: традиционный итератор и сплитератор, появившийся в версии JDK 8, который обеспечивает значительные преимущества в определенных ситуациях использования параллельных потоков.

Использование итератора с потоком

Как уже упоминалось, применять итератор с потоком можно таким же способом, как с коллекцией. Итераторы обсуждались в главе 20, но здесь полезно кратко напомнить о них. Итераторы — это объекты, реализующие интерфейс Iterator из пакета java.util. Двумя ключевыми методами являются hasNext() и next(). Метод hasNext() возвращает true при наличии еще одного элемента для итерации или false в противном случае. Метод next() возвращает следующий элемент в итерации.

На заметку! Существуют дополнительные типы итераторов, которые обрабатывают потоки примитивных типов: PrimitiveIterator, PrimitiveIterator.OfDouble, PrimitiveIterator.OfLong и PrimitiveIterator.OfInt. Все они расширяют интерфейс Iterator и работают аналогично тем, которые основаны непосредственно на Iterator.

Чтобы получить итератор для потока, понадобится вызвать метод `iterator()` класса `Stream`:

```
Iterator<T> iterator()
```

В `T` указывается тип элементов. (Потоки примитивных типов возвращают итераторы соответствующих примитивных типов.)

В приведенной далее программе демонстрируется поход по элементам потока. В данном случае реализована итерация по строкам в `ArrayList`, но процесс одинаков для любого типа потока:

```
// Использование итератора с потоком.

import java.util.*;
import java.util.stream.*;

class StreamDemo8 {
  public static void main(String[] args) {
    // Создать список строк.
    ArrayList<String> myList = new ArrayList<>();
    myList.add("Alpha");
    myList.add("Beta");
    myList.add("Gamma");
    myList.add("Delta");
    myList.add("Phi");
    myList.add("Omega");

    // Получить поток для ArrayList.
    Stream<String> myStream = myList.stream();

    // Получить итератор для потока.
    Iterator<String> itr = myStream.iterator();

    // Пройти по элементам в потоке.
    while(itr.hasNext())
      System.out.println(itr.next());
  }
}
```

Ниже показан вывод:

```
Alpha
Beta
Gamma
Delta
Phi
Omega
```

Использование сплитератора

Сплитератор — это альтернатива итератору, особенно когда задействована параллельная обработка. В целом сплитератор сложнее итератора, а обсуждение сплитераторов можно найти в главе 20, но полезно вспомнить об его основных возможностях. В интерфейсе `Spliterator` определено несколько методов, но здесь интересуют только три из них. Первый — `tryAdvance()`, который выполняет действие со следующим элементом и затем перемещает итератор:

```
boolean tryAdvance(Consumer<? super T> action)
```

В action указывается действие, которое выполняется для следующего элемента в итерации. Метод tryAdvance() возвращает true при наличии следующего элемента или false, если элементов не осталось. Как обсуждалось ранее в главе, в интерфейсе Consumer объявлен один метод по имени accept(), который получает элемент типа T в качестве аргумента и возвращает значение void.

Поскольку метод tryAdvance() возвращает false, когда больше нет элементов для обработки, конструкция цикла итерации становится очень простой, например:

```
while(splitItr.tryAdvance(
  // выполнение действия
);
```

До тех пор пока tryAdvance() возвращает true, действие применяется к следующему элементу. Когда tryAdvance() возвращает false, итерация завершается. Обратите внимание на то, что tryAdvance() объединяет функциональность hasNext() и next(), предоставляемые Iterator, в один метод, увеличивая эффективность самого процесса итерации.

В следующей версии предыдущей программы вместо итератора используется сплитератор:

```
// Использование сплитератора.

import java.util.*;
import java.util.stream.*;

class StreamDemo9 {
  public static void main(String[] args) {

    // Создать список строк.
    ArrayList<String> myList = new ArrayList<>();
    myList.add("Alpha");
    myList.add("Beta");
    myList.add("Gamma");
    myList.add("Delta");
    myList.add("Phi");
    myList.add("Omega");

    // Получить поток для ArrayList.
    Stream<String> myStream = myList.stream();

    // Получить сплитератор.
    Spliterator<String> splitItr = myStream.spliterator();

    // Пройти по элементам в потоке.
    while(splitItr.tryAdvance((n) -> System.out.println(n)));
  }
}
```

Вывод остается прежним.

В ряде случаев может требоваться выполнение какого-то действия над всеми, а не отдельными элементами. Справиться с такой ситуацией позволяет метод forEachRemaining(), предоставляемый интерфейсом Spliterator:

```
default void forEachRemaining(Consumer<? super T> action)
```

Метод forEachRemaining() применяет действие к каждому необработанному элементу, после чего возвращает значение. Например, продолжая предыдущую программу, следующий код отображает строки, оставшиеся в потоке:

```
splitItr.forEachRemaining((n) -> System.out.println(n));
```

Обратите внимание, что данный метод устраняет необходимость в цикле для последовательного прохода по элементам — это еще одно преимущество сплитератора.

Интересен еще один метод интерфейса Spliterator — trySplit(). Он разбивает итерируемые элементы на две части, возвращая новый сплитератор для одной из частей. Другая часть остается доступной для первоначального сплитератора:

```
Spliterator<T> trySplit()
```

Если вызывающий сплитератор разделить невозможно, тогда возвращается null, а иначе ссылка на часть. Например, вот еще одна версия предыдущей программы, демонстрирующая работу метода trySplit():

```
// Демонстрация использования trySplit().

import java.util.*;
import java.util.stream.*;

class StreamDemo10 {
  public static void main(String[] args) {
    // Создать список строк.
    ArrayList<String> myList = new ArrayList<>();
    myList.add("Alpha");
    myList.add("Beta");
    myList.add("Gamma");
    myList.add("Delta");
    myList.add("Phi");
    myList.add("Omega");

    // Получить поток для ArrayList.
    Stream<String> myStream = myList.stream();

    // Получить сплитератор.
    Spliterator<String> splitItr = myStream.spliterator();

    // Разбить на части splitItr.
    Spliterator<String> splitItr2 = splitItr.trySplit();

    // Если splitItr не удается разбить на части,
    // тогда сначала использовать splitItr2.
    if(splitItr2 != null) {
      System.out.println("Вывод из splitItr2: ");
      splitItr2.forEachRemaining((n) -> System.out.println(n));
    }

    // Теперь использовать splitItr.
    System.out.println("\nВывод из splitItr: ");
    splitItr.forEachRemaining((n) -> System.out.println(n));
  }
}
```

Вот вывод, генерируемый программой:

```
Вывод из splitItr2:
Alpha
Beta
Gamma

Вывод из splitItr:
Delta
Phi
Omega
```

Хотя разделение сплитератора в этом простом примере не имеет практической ценности, оно может быть крайне важным при параллельной обработке крупных наборов данных. Тем не менее, во многих случаях лучше использовать какой-то другой метод потока в сочетании с параллельным потоком, а не вручную обрабатывать такие детали с помощью сплитератора, который предназначен в первую очередь для ситуаций, когда ни один из предопределенных методов не выглядит подходящим.

Дальнейшее исследование потокового API

В главе обсуждалось несколько ключевых аспектов потокового API и были представлены способы их применения, но потоковый API способен предложить гораздо больше. Для начала вот несколько других методов, предоставляемых классом `Stream`, которые вы сочтете полезными:

- для выяснения, удовлетворяют ли один или несколько элементов в потоке указанному предикату, используются методы `allMatch()`, `anyMatch()` и `noneMatch()`;
- для получения количества элементов в потоке применяется метод `count()`;
- для получения потока, который содержит только уникальные элементы, используется метод `distinct()`;
- для создания потока, содержащего указанный набор элементов, применяется метод `of()`.

Наконец, важно отметить, что потоковый API является мощным аспектом языка Java. Рекомендуется изучить все возможности, которые предлагает пакет `java.util.stream`.

ГЛАВА

31 Регулярные выражения и другие пакеты

В состав первоначального выпуска Java входил набор из восьми пакетов, называемых *основным API*. Каждый последующий выпуск что-то добавлял в API. На сегодняшний день Java API содержит очень большое количество пакетов. Многие пакеты поддерживают области специализации, выходящие за рамки настоящей книги, но некоторые пакеты требуют краткого введения здесь. Четырьмя из них являются `java.util.regex`, `java.lang.reflect`, `java.rmi` и `java.text`. Они поддерживают соответственно обработку регулярных выражений, рефлексию, удаленный вызов методов и форматирование текста. В конце главы дается введение в API даты и времени из пакета `java.time` и его подпакетов.

Пакет *регулярных выражений* позволяет выполнять сложные операции сопоставления с шаблоном. В главе представлено введение в данный пакет вместе с подробными примерами. *Рефлексия* — это способность программного обеспечения анализировать себя и неотъемлемая часть технологии Java Beans, которая рассматривается в главе 35. *Удаленный вызов методов* (Remote Method Invocation — RMI) позволяет создавать приложения Java, распределенные между несколькими машинами. В главе приведен простой пример клиент-серверного приложения, использующий RMI. Возможности *форматирования текста* в `java.text` имеют множество применений. Здесь рассматривается форматирование строк с датой и временем. API даты и времени обеспечивает современный подход к обработке даты и времени.

Обработка регулярных выражений

Пакет `java.util.regex` поддерживает обработку регулярных выражений. Начиная с версии JDK 9, пакет `java.util.regex` находится в модуле `java.base`. Под *регулярным выражением* здесь понимается строка, описывающая последовательность символов. Такое общее описание, называемое шаблоном, затем можно использовать для поиска совпадений в других последовательностях символов. В регулярных выражениях можно указывать групповые символы, наборы символов и различные квантификаторы. Таким образом, можно задать регулярное выражение, представляющее общую форму, которая может соответствовать нескольким конкретным последовательностям символов.

Обработку регулярных выражений поддерживают два класса: Pattern и Matcher. Они работают вместе. Класс Pattern применяется для определения регулярного выражения, а класс Matcher — для сопоставления шаблона с другой последовательностью.

Класс Pattern

Конструкторы в классе Pattern не определены. Взамен шаблон создается путем вызова фабричного метода compile(). Вот одна из его форм:

```
static Pattern compile(String pattern)
```

В pattern передается регулярное выражение, которое необходимо использовать. Метод compile() преобразует строку в шаблоне в шаблон, который может применяться для сопоставления с шаблоном через класс Matcher. Метод возвращает объект Pattern, содержащий шаблон.

Построенный объект Pattern будет использоваться для создания объекта Matcher, для чего вызывается метод matcher(), определенный в Pattern:

```
Matcher matcher(CharSequence str)
```

В str указывается последовательность символов, с которой будет сопоставляться шаблон. Она называется *входной последовательностью*. Интерфейс CharSequence определяет доступный только для чтения набор символов и реализуется среди прочих классом String. Таким образом, методу matcher() можно передавать строку.

Класс Matcher

Класс Matcher не имеет конструкторов. Объект Matcher создается вызовом фабричного метода matcher(), определенного в Pattern. Методы созданного объекта Matcher будут применяться для выполнения различных операций сопоставления с шаблоном. Ниже описано несколько таких методов.

Простейшим методом сопоставления с шаблоном является match(), который выясняет, совпадает ли последовательность символов с шаблоном:

```
boolean matches()
```

Метод match() возвращает true, если последовательность символов совпадает с шаблоном, или false в противном случае. Важно понимать, что совпадать с шаблоном должна вся последовательность, а не только ее часть.

Для выяснения, совпадает ли с шаблоном какая-то подпоследовательность входной последовательности, предназначен метод find(). Ниже показана одна из его форм:

```
boolean find()
```

Он возвращает true, если есть совпадающая подпоследовательность, или false в противном случае. Данный метод можно вызывать многократно, что позволяет отыскать все совпадающие подпоследовательности. Каждый последующий вызов find() начинает работу с того места, где ее закончил предыдущий.

Вызвав метод group(), можно получить строку, содержащую последнюю совпавшую последовательность. Вот одна из его форм:

```
String group()
```

Метод group() возвращает совпавшую строку. Если совпадений не было обнаружено, тогда генерируется исключение IllegalStateException.

Индекс текущего совпадения во входной последовательности можно получить с помощью метода start(), а индекс, следующий за концом текущего совпадения — посредством метода end(). Далее в главе используются следующие формы упомянутых методов:

```
int start()
int end()
```

Оба метода генерируют исключение IllegalStateException, если совпадений не обнаружено.

Вызвав метод replaceAll(), одна из форм которого показана ниже, можно заменить все вхождения совпадающей последовательности другой последовательностью:

```
String replaceAll(String newStr)
```

В newStr указывается новая последовательность символов, которая заменит последовательность, совпадающую с шаблоном. Метод возвращает обновленную входную последовательность в виде строки.

Синтаксис регулярных выражений

Прежде чем демонстрировать применение классов Pattern и Matcher, необходимо выяснить, как строить регулярное выражение. Хотя ни одно правило само по себе не является сложным, их существует большое количество, и подробное обсуждение всех правил выходит за рамки главы. Однако здесь описаны некоторые наиболее часто используемые конструкции.

Обычно регулярное выражение состоит из нормальных символов, классов символов (наборов символов), групповых знаков и квантификаторов. Нормальный символ сопоставляется в том виде как есть. Таким образом, если шаблон состоит из "xy", то единственной входной последовательностью, которая совпадет с ним, будет "xy". Символы вроде новой строки и табуляции указываются с применением стандартных управляющих последовательностей, начинающихся с символа \ . Например, новая строка определяется как \n. На языке регулярных выражений нормальный символ также называется *литералом*.

Класс символов представляет собой набор символов и указывается путем помещения символов класса в квадратные скобки. Например, класс [wxyz] дает совпадение с символом w, x, y или z. Чтобы задать обратный набор, перед символами понадобится указать ^. Например, [^wxyz] совпадает с любым символом кроме w, x, y или z. С помощью дефиса можно задать диапазон символов. Например, чтобы указать класс символов, который будет совпадать с цифрами от 1 до 9, необходимо использовать [1-9].

Групповой символ обозначается посредством точки (.) и соответствует любому символу. Таким образом, шаблон, состоящий из ".", будет совпадать со следующими входными последовательностями: "A", "a", "x" и т.д.

Квантификаторы определяют, сколько раз совпадает выражение.

В табл. 31.1 кратко описаны базовые квантификаторы.

Таблица 31.1. Базовые квантификаторы

Квантификатор	Описание
+	Совпадение один или больше раз
*	Совпадение ноль или больше раз
?	Совпадение ноль или один раз

Например, шаблон "x+" совпадет среди прочего с "x", "xx" и "xxx". Как будет показано, поддерживаются вариации, которые влияют на способ сопоставления.

И еще один момент: в общем случае при указании недопустимого выражения сгенерируется исключение PatternSyntaxException.

Демонстрация сопоставления с шаблоном

Понять работу сопоставления с шаблоном регулярных выражений лучше всего, рассмотрев несколько примеров. В первом примере ищется совпадение с литеральным шаблоном:

```
// Простая демонстрация работы сопоставления с шаблоном.
import java.util.regex.*;

class RegExpr {
  public static void main(String[] args) {
    Pattern pat;
    Matcher mat;
    boolean found;

    pat = Pattern.compile("Java");
    mat = pat.matcher("Java");
    found = mat.matches();              // проверить на совпадение

    System.out.println("Проверка на совпадение Java с Java.");
    if(found) System.out.println("Совпадает");
    else System.out.println("Не совпадает");

    System.out.println();

    System.out.println("Проверка на совпадение Java с Java SE.");
    mat = pat.matcher("Java SE");       // создать новый объект Matcher

    found = mat.matches();              // проверить на совпадение

    if(found) System.out.println("Совпадает");
    else System.out.println("Не совпадает");
  }
}
```

Вот вывод, генерируемый программой:

```
Проверка на совпадение Java с Java.
Совпадает

Проверка на совпадение Java с Java SE.
Не совпадает
```

Давайте проанализируем программу. Сначала создается шаблон, содержащий последовательность "Java". Затем для шаблона создается объект Matcher, который имеет входную последовательность "Java". Далее вызывается метод match(), чтобы выяснить, совпадает ли входная последовательность с шаблоном. Поскольку последовательность и шаблон одинаковы, метод match() возвращает true. Наконец, создается новый объект Matcher с входной последовательностью "Java SE" и снова вызывается метод match(). В данном случае шаблон и входная последовательность отличаются, а потому совпадение не найдено. Не забывайте, что метод match() возвращает true только в ситуации, когда входная последовательность точно соответствует шаблону. Он не возвратит true лишь потому, что совпадает какая-то часть последовательности.

С помощью метода find() можно определить, содержит ли входная последовательность подпоследовательность, которая совпадает с шаблоном. Взгляните на следующую программу:

```
// Использование метода find() для поиска подпоследовательности.

import java.util.regex.*;

class RegExpr2 {
  public static void main(String[] args) {
    Pattern pat = Pattern.compile("Java");
    Matcher mat = pat.matcher("Java SE");

    System.out.println("Поиск Java в Java SE.");

    if(mat.find()) System.out.println("Подпоследовательность найдена");
    else System.out.println("Совпадений не обнаружено");
  }
}
```

Ниже показан вывод:

```
Поиск Java в Java SE.
Подпоследовательность найдена
```

В данном случае метод find() отыскал подпоследовательность "Java".

Метод find() можно применять для поиска повторяющихся вхождений шаблона во входной последовательности, т.к. каждый последующий вызов find() начинает работу с того места, где остановился предыдущий. Скажем, в приведенной далее программе ищутся два вхождения шаблона "test":

```
// Использование метода find() для поиска нескольких вхождений.

import java.util.regex.*;

class RegExpr3 {
  public static void main(String[] args) {
```

```
    Pattern pat = Pattern.compile("тест");
    Matcher mat = pat.matcher("тест 1 2 3 тест");

    while(mat.find()) {
      System.out.println("тест найдено по индексу " +
      mat.start());
    }
  }
}
```

Вот вывод, генерируемый программой:

```
тест найдено по индексу 0
тест найдено по индексу 11
```

В выводе видно, что обнаружено два совпадения. Для получения индекса каждого совпадения используется метод `start()`.

Использование групповых символов и квантификаторов

Хотя в предшествующих программах была показана общая методика применения классов `Pattern` и `Matcher`, там не демонстрировались все их возможности. Реальная польза от обработки регулярных выражений не будет заметна до тех пор, пока не начнут использоваться групповые символы и квантификаторы. Первым делом рассмотрим пример, в котором посредством квантификатора + обеспечивается сопоставление с любой произвольно длинной последовательностью символов `W`:

```
// Использование квантификатора.

import java.util.regex.*;

class RegExpr4 {
  public static void main(String[] args) {
    Pattern pat = Pattern.compile("W+");
    Matcher mat = pat.matcher("W WW WWW");

    while(mat.find())
    System.out.println("Совпадение: " + mat.group());
  }
}
```

Вывод из программы выглядит следующим образом:

```
Совпадение: W
Совпадение: WW
Совпадение: WWW
```

Легко заметить, что шаблон с регулярным выражением `"W+"` дает совпадение с любой произвольно длинной последовательностью символов `W`.

В приведенной ниже программе групповой символ `.` применяется вместе с квантификатором + для создания шаблона, который будет соответствовать любой последовательности, начинающейся с *e* и заканчивающейся на *d*:

```
// Использование группового символа и квантификатора.

import java.util.regex.*;
```

```
class RegExpr5 {
  public static void main(String[] args) {
    Pattern pat = Pattern.compile("e.+d");
    Matcher mat = pat.matcher("extend cup end table");

    while(mat.find())
      System.out.println("Совпадение: " + mat.group());
  }
}
```

Вывод, генерируемый программой, может удивить:

```
Совпадение: extend cup end
```

Обнаружено только одно совпадение — самая длинная последовательность, начинающаяся с *e* и заканчивающаяся на *d*. Можно было бы ожидать два совпадения: "extend" и "end". Причина того, что найдена более длинная последовательность, связана с тем, что шаблон "e.+d" соответствует самой длинной последовательности, удовлетворяющей шаблону. Такое поведение называется *жадным*. Добавив символ ? к шаблону, как показано в следующей версии программы, можно задать *ленивое* поведение, что приведет к получению кратчайшего совпадения с шаблоном:

```
// Использование квантификатора ленивого поведения.

import java.util.regex.*;

class RegExpr6 {
  public static void main(String[] args) {
    // Задать ленивое поведение сопоставления.
    Pattern pat = Pattern.compile("e.+?d");
    Matcher mat = pat.matcher("extend cup end table");

    while(mat.find())
      System.out.println("Совпадение: " + mat.group());
  }
}
```

Вот вывод из программы:

```
Совпадение: extend
Совпадение: end
```

В выводе видно, что шаблон "e.+?d" дает совпадение с кратчайшей последовательностью, начинающейся с *e* и заканчивающейся на *d*. В итоге обнаруживаются два совпадения.

В общем случае для преобразования жадного квантификатора в ленивый необходимо добавить ?. Также можно задать собственническое поведение, добавив символ +. В качестве примера опробуйте шаблон "e.?+d" и взгляните на результат. Кроме того, можно указать количество совпадений с использованием конструкции {min, limit}, обеспечивающей совпадение от min до limit раз. Также поддерживаются варианты {min} и {min,}, которые соответственно дадут совпадения min раз и min раз, но возможно больше.

Работа с классами символов

Временами может требоваться сопоставление с любой последовательностью, содержащей один или несколько символов в любом порядке, которые являются частью набора символов. Скажем, чтобы сопоставлять с полными словами, понадобится организовать сопоставление с любой последовательностью букв алфавита. Один из самых простых способов добиться цели — применить класс символов, который определяет набор символов. Вспомните, что класс символов создается путем помещения в квадратные скобки символов, подлежащих сопоставлению. Например, для сопоставления символов нижнего регистра от *a* до *z* необходимо использовать `[a-z]`. Прием демонстрируется в следующей программе:

```
// Использование класса символов.
import java.util.regex.*;
class RegExpr7 {
  public static void main(String[] args) {
    // Сопоставлять со словами в нижнем регистре.
    Pattern pat = Pattern.compile("[a-z]+");
    Matcher mat = pat.matcher("this is a test.");
    while(mat.find())
      System.out.println("Совпадение: " + mat.group());
  }
}
```

Ниже показан вывод, генерируемый программой:

```
Совпадение: this
Совпадение: is
Совпадение: a
Совпадение: test
```

Использование метода `replaceAll()`

Метод `replaceAll()`, предоставляемый классом `Matcher`, позволяет выполнять мощные операции поиска и замены с применением регулярных выражений. Скажем, в приведенной далее программе все вхождения последовательностей, начинающихся с `"Jon"`, заменяются на `"Eric"`:

```
// Использование метода replaceAll().
import java.util.regex.*;
class RegExpr8 {
  public static void main(String[] args) {
    String str = "Jon Jonathan Frank Ken Todd";

    Pattern pat = Pattern.compile("Jon.*? ");
    Matcher mat = pat.matcher(str);

    System.out.println("Исходная последовательность: " + str);
    str = mat.replaceAll("Eric ");
    System.out.println("Модифицированная последовательность: " + str);
  }
}
```

Вот вывод из программы:

```
Исходная последовательность: Jon Jonathan Frank Ken Todd
Модифицированная последовательность: Eric Eric Frank Ken Todd
```

Поскольку регулярное выражение "Jon.*?" соответствует любой строке, начинающейся с Jon, за которой следует ноль или более символов, а в конце пробел, его можно использовать для сопоставления и замены Jon и Jonathan именем Eric. Такую замену нелегко выполнить без возможностей сопоставления с шаблоном.

Использование метода split()

С помощью метода split(), определенного в Pattern, входную последовательность можно преобразовать в отдельные лексемы. Ниже показана одна из форм метода split():

```
String[] split(CharSequence str)
```

Он обрабатывает переданную в str входную последовательность, выдавая лексемы на основе заданных шаблоном разделителей.

Например, в следующей программе ищутся лексемы, разделенные пробелами, запятыми, точками и восклицательными знаками:

```
// Использование метода split().
import java.util.regex.*;
class RegExpr9 {
  public static void main(String[] args) {
    // Сопоставить со словами в нижнем регистре.
    Pattern pat = Pattern.compile("[ ,.!]");
    String[] strs = pat.split("one two,alpha9 12!done.");
    for(int i=0; i < strs.length; i++)
      System.out.println("Следующая лексема: " + strs[i]);
  }
}
```

Вот вывод:

```
Следующая лексема: one
Следующая лексема: two
Следующая лексема: alpha9
Следующая лексема: 12
Следующая лексема: done
```

В выводе видно, что входная последовательность преобразуется в отдельные лексемы. Обратите внимание, что разделители не включаются.

Два варианта сопоставления с шаблоном

Несмотря на то что описанные выше методики сопоставления с шаблоном обеспечивают наибольшую гибкость и эффективность, есть две альтернативы, которые в некоторых обстоятельствах могут оказаться полезными. Если нужно выполнить только однократное сопоставление с шаблоном, тогда можно применить метод matches(), определенный в Pattern:

```
static boolean matches(String pattern, CharSequence str)
```

Он возвращает true, если шаблон pattern совпадает со строкой str, или false в противном случае. Метод matches() автоматически компилирует шаблон, после чего ищет совпадение. При многократном использовании одного и того же шаблона метод matches() будет менее эффективным, чем компиляция шаблона и применение методов сопоставления с шаблоном, определенных в классе Matcher.

Выполнить сопоставление с шаблоном можно также с использованием метода matches(), реализованного в String:

```
boolean matches(String pattern)
```

Если вызывающая строка дает совпадение с регулярным выражением в pattern, то метод matches() возвращает true. Иначе он возвращает false.

Дальнейшее исследование регулярных выражений

Обзор регулярных выражений, представленный в настоящем разделе, дает лишь поверхностное представление об их истинных возможностях. Поскольку синтаксический разбор, манипулирование и разбиение на лексемы текста — часто встречающиеся задачи программирования, подсистема регулярных выражений Java наверняка покажется вам мощным инструментом, который можно задействовать в своих интересах. Поэтому разумно исследовать все возможности регулярных выражений. Поэкспериментируйте с несколькими типами шаблонов и входных последовательностей. Хорошо разобравшись в том, как работает сопоставление с шаблонами в форме регулярных выражений, вы сочтете его полезным при решении многих задач программирования.

Рефлексия

Под рефлексией понимается способность программного обеспечения анализировать себя, которая обеспечивается пакетом java.lang.reflect и элементами в классе Class. Начиная с версии JDK 9, пакет java.lang.reflect является частью модуля java.base. Рефлексия — важное средство, особенно при работе с компонентами Java Beans. Она позволяет анализировать программный компонент и динамически описывать его возможности во время выполнения, а не на этапе компиляции. Скажем, с применением рефлексии можно выяснить, какие методы, конструкторы и поля поддерживает класс. Рефлексия была представлена в главе 12, а здесь она рассматривается более подробно.

Пакет java.lang.reflect включает несколько интерфейсов. Особый интерес представляет Member, в котором определены методы, позволяющие получить информацию о поле, конструкторе или методе класса. Кроме того, пакет java.lang.reflect содержит 11 классов, которые кратко описаны в табл. 31.2.

Таблица 31.2. Классы, определенные в пакете `java.lang.reflect`

Класс	Описание
`AccessibleObject`	Позволяет обходить стандартные проверки управления доступом
`Array`	Позволяет динамически создавать массивы и манипулировать ими
`Constructor`	Предоставляет информацию о конструкторе
`Executable`	Абстрактный суперкласс, расширяемый классами `Method` и `Constructor`
`Field`	Предоставляет информацию о поле
`Method`	Предоставляет информацию о методе
`Modifier`	Предоставляет информацию о модификаторах доступа к классу и его членам
`Parameter`	Предоставляет информацию о параметрах
`Proxy`	Поддерживает динамические классы-посредники
`RecordComponent`	Предоставляет информацию о записи
`ReflectPermission`	Делает возможной рефлексию закрытых или защищенных членов класса

В следующей программе иллюстрируется простое использование возможностей рефлексии Java. Она выводит конструкторы, поля и методы класса `java.awt.Dimension`. Сначала с применением метода `forName()` класса `Class` получается объект `Class` для `java.awt.Dimension`, после чего этот объект анализируется с использованием методов `getConstructors()`, `getFields()` и `getMethods()`. Упомянутые методы возвращают массивы объектов `Constructor`, `Field` и `Method`, предоставляющие информацию об объекте. В классах `Constructor`, `Field` и `Method` определено несколько методов, которые можно применять для получения сведений об объекте. Рекомендуется изучить их самостоятельно. Следует отметить, что каждый из них поддерживает метод `toString()`. Таким образом, использовать объекты `Constructor`, `Field` и `Method` в качестве аргументов метода `println()` очень легко.

```
// Демонстрация применения рефлексии.
import java.lang.reflect.*;
public class ReflectionDemo1 {
  public static void main(String[] args) {
    try {
      Class<?> c = Class.forName("java.awt.Dimension");
      System.out.println("Конструкторы:");
```

```
      Constructor<?>[] constructors = c.getConstructors();
      for(int i = 0; i < constructors.length; i++) {
        System.out.println(" " + constructors[i]);
      }
      System.out.println("Поля:");
      Field[] fields = c.getFields();
      for(int i = 0; i < fields.length; i++) {
        System.out.println(" " + fields[i]);
      }
      System.out.println("Методы:");
      Method[] methods = c.getMethods();
      for(int i = 0; i < methods.length; i++) {
        System.out.println(" " + methods[i]);
      }
    }
    catch(Exception e) {
      System.out.println("Возникло исключение " + e);
    }
  }
}
```

Ниже показан вывод, генерируемый программой, который у вас может слегка отличаться:

```
Конструкторы:
 public java.awt.Dimension(int,int)
 public java.awt.Dimension()
 public java.awt.Dimension(java.awt.Dimension)
Поля:
 public int java.awt.Dimension.width
 public int java.awt.Dimension.height
Методы:
 public int java.awt.Dimension.hashCode()
 public boolean java.awt.Dimension.equals(java.lang.Object)
 public java.lang.String java.awt.Dimension.toString()
 public java.awt.Dimension java.awt.Dimension.getSize()
 public void java.awt.Dimension.setSize(double,double)
 public void java.awt.Dimension.setSize(java.awt.Dimension)
 public void java.awt.Dimension.setSize(int,int)
 public double java.awt.Dimension.getHeight()
 public double java.awt.Dimension.getWidth()
 public java.lang.Object java.awt.geom.Dimension2D.clone()
 public void java.awt.geom.Dimension2D.setSize(java.awt.geom.Dimension2D)
 public final native java.lang.Class java.lang.Object.getClass()
 public final native void java.lang.Object.wait(long)
   throws java.lang.InterruptedException
 public final void java.lang.Object.wait()
   throws java.lang.InterruptedException
 public final void java.lang.Object.wait(long,int)
   throws java.lang.InterruptedException
 public final native void java.lang.Object.notify()
 public final native void java.lang.Object.notifyAll()
```

В приведенном далее примере используются возможности рефлексии Java для получения открытых методов класса. Программа начинается с создания экземпляра класса A. К этой объектной ссылке применяется метод getClass(), который возвращает объект Class для класса A. Метод getDeclaredMethods() возвращает массив объектов Method, описывающий только методы, которые объявлены в этом классе. Методы, унаследованные от суперклассов вроде Object, не включаются.

Затем обрабатывается каждый элемент массива methods. Метод getModifiers() возвращает значение int с флагами, описывающими модификаторы, которые применяются к данному элементу. Класс Modifier предоставляет набор методов isX(), кратко описанных в табл. 31.3, которые можно использовать для проверки этого значения. Например, статический метод isPublic() возвращает true, если его аргумент включает модификатор public, или false в противном случае.

Таблица 31.3. Методы isX() класса Modifier, определяющие модификаторы

Метод	Описание
static boolean isAbstract(int val)	Возвращает true, если val имеет установленный флаг abstract, или false в противном случае
static boolean isFinal(int val)	Возвращает true, если val имеет установленный флаг final, или false в противном случае
static boolean isInterface(int val)	Возвращает true, если val имеет установленный флаг interface, или false в противном случае
static boolean isNative(int val)	Возвращает true, если val имеет установленный флаг native, или false в противном случае
static boolean isPrivate(int val)	Возвращает true, если val имеет установленный флаг private, или false в противном случае
static boolean isProtected(int val)	Возвращает true, если val имеет установленный флаг protected, или false в противном случае
static boolean isPublic(int val)	Возвращает true, если val имеет установленный флаг public, или false в противном случае
static boolean isStatic(int val)	Возвращает true, если val имеет установленный флаг static, или false в противном случае
static boolean isStrict(int val)	Возвращает true, если val имеет установленный флаг strict, или false в противном случае
static boolean isSynchronized(int val)	Возвращает true, если val имеет установленный флаг synchronized, или false в противном случае
static boolean isTransient(int val)	Возвращает true, если val имеет установленный флаг transient, или false в противном случае
static boolean isVolatile(int val)	Возвращает true, если val имеет установленный флаг volatile, или false в противном случае

В следующей программе сначала посредством вызова getName() получаются имена открытых методов, которые затем выводятся:

```
// Отображение открытых методов.
import java.lang.reflect.*;
public class ReflectionDemo2 {
  public static void main(String[] args) {
    try {
      A a = new A();
      Class<?> c = a.getClass();
      System.out.println("Открытые методы:");
      Method[] methods = c.getDeclaredMethods();
      for(int i = 0; i < methods.length; i++) {
        int modifiers = methods[i].getModifiers();
        if(Modifier.isPublic(modifiers)) {
          System.out.println(" " + methods[i].getName());
        }
      }
    }
    catch(Exception e) {
      System.out.println("Возникло исключение " + e);
    }
  }
}
class A {
  public void a1() {
  }
  public void a2() {
  }
  protected void a3() {
  }
  private void a4() {
  }
}
```

Вот вывод:

```
Открытые методы:
a1
a2
```

В классе Modifier также есть набор статических методов, возвращающих тип модификаторов, которые можно применить к специфическому типу элемента программы:

```
static int classModifiers()
static int constructorModifiers()
static int fieldModifiers()
static int interfaceModifiers()
static int methodModifiers()
static int parameterModifiers()
```

Скажем, methodModifiers() возвращает модификаторы, которые можно применить к методу. Каждый метод возвращает упакованные в значение int

флаги, указывающие допустимые модификаторы. Значения модификаторов определяются константами в классе Modifier: PROTECTED, PUBLIC, PRIVATE, STATIC, FINAL и т.д.

Удаленный вызов методов

Механизм удаленного вызова методов (RMI) позволяет объекту Java, который функционирует на одной машине, вызывать метод объекта Java, выполняющегося на другой машине. Важность средства RMI в том, что оно позволяет строить распределенные приложения. Хотя подробное обсуждение RMI выходит за рамки книги, в рассмотренном ниже упрощенном примере демонстрируются базовые принципы. Удаленный вызов методов поддерживается пакетом java.rmi, который в версии JDK 9 стал частью модуля java.rmi.

Простое клиент-серверное приложение, использующее удаленный вызов методов

В этом разделе представлены пошаговые инструкции по созданию простого клиент-серверного приложения, использующего средство RMI. Сервер получает запрос от клиента, обрабатывает его и возвращает результат. Здесь в запросе указываются два числа. Сервер складывает их и возвращает сумму.

Шаг 1: подготовка и компиляция исходного кода

В приложении задействованы четыре файла исходного кода. В первом файле, AddServerIntf.java, определен удаленный интерфейс, предоставляемый сервером, который содержит один метод, принимающий два аргумента типа double и возвращающий их сумму. Все удаленные интерфейсы должны расширять интерфейс Remote, который входит в состав java.rmi. В интерфейсе Remote члены не определены. Его цель — просто указать, что интерфейс применяет удаленные методы. Все удаленные методы могут генерировать исключение RemoteException.

```
import java.rmi.*;

public interface AddServerIntf extends Remote {
  double add(double d1, double d2) throws RemoteException;
}
```

Во втором файле исходного кода, AddServerImpl.java, реализован удаленный интерфейс. Реализация метода add() проста. Удаленные классы обычно расширяют класс UnicastRemoteObject, который предоставляет функциональные возможности, необходимые для обеспечения доступности объектов из удаленных машин.

```
import java.rmi.*;
import java.rmi.server.*;

public class AddServerImpl extends UnicastRemoteObject implements
  AddServerIntf {
```

```
  public AddServerImpl() throws RemoteException {
  }
  public double add(double d1, double d2) throws RemoteException {
    return d1 + d2;
  }
}
```

Третий файл исходного кода, `AddServer.java`, содержит главную программу для сервера. Его основная функция — обновление реестра RMI на данной машине, что делается с помощью метода `rebind()` класса `Naming` (из `java.rmi`), который связывает имя с объектной ссылкой. Первым аргументом метода `rebind()` является строка, именующая сервер как `"AddServer"`, а вторым аргументом — ссылка на экземпляр `AddServerImpl`.

```
import java.net.*;
import java.rmi.*;

public class AddServer {
  public static void main(String[] args) {
    try {
      AddServerImpl addServerImpl = new AddServerImpl();
      Naming.rebind("AddServer", addServerImpl);
    }
    catch(Exception e) {
      System.out.println("Возникло исключение " + e);
    }
  }
}
```

В четвертом файле исходного кода, `AddClient.java`, реализована клиентская часть распределенного приложения. Для `AddClient.java` требуются три аргумента командной строки. Первый — IP-адрес или имя сервера, а второй и третий — два числа, которые необходимо просуммировать.

Приложение начинается с формирования строки в соответствии с синтаксисом URL. Результирующий URL использует протокол `rmi`. Строка включает IP-адрес либо имя сервера и строку `"AddServer"`. Затем вызывается метод `lookup()` класса `Naming`, который принимает один аргумент, URL-адрес `rmi`, и возвращает ссылку на объект типа `AddServerIntf`. Все удаленные вызовы методов можно будет направлять данному объекту.

Приложение продолжается отображением своих аргументов и вызовом метода `add()`, возвращающего сумму, которая затем выводится.

```
import java.rmi.*;

public class AddClient {
  public static void main(String[] args) {
    try {
      String addServerURL = "rmi://" + args[0] + "/AddServer";
      AddServerIntf addServerIntf
            = (AddServerIntf)Naming.lookup(addServerURL);
      System.out.println("Первое число: " + args[1]);
      double d1 = Double.valueOf(args[1]).doubleValue();
      System.out.println("Второе число: " + args[2]);
```

```
      double d2 = Double.valueOf(args[2]).doubleValue();
      System.out.println("Сумма: " + addServerIntf.add(d1, d2));
    }
    catch(Exception e) {
      System.out.println("Возникло исключение " + e);
    }
  }
}
```

После создания всех файлов исходного кода их нужно скомпилировать с помощью `javac`.

Шаг 2: создание заглушки вручную, если она необходима

В контексте RMI *заглушка* — это объект Java, который находится на клиентской машине. Его функция заключается в том, чтобы представлять те же интерфейсы, что и удаленный сервер. Вызовы удаленных методов, инициированные клиентом, фактически направляются заглушке. Заглушка взаимодействует с другими частями системы RMI для формирования запроса, который отправляется удаленной машине.

Удаленный метод может принимать аргументы, относящиеся к простым типам или объектам. В последнем случае объект может ссылаться на другие объекты. Вся информация должна быть отправлена удаленной машине, т.е. объект, переданный в качестве аргумента при вызове удаленного метода, должен быть сериализирован и передан на удаленную машину. Вспомните из главы 21, что средства сериализации рекурсивно обрабатывают все объекты, на которые имеются ссылки.

Если ответ должен быть возвращен клиенту, тогда процесс работает в обратном порядке. Обратите внимание, что при возвращении клиенту объектов также применяются средства сериализации и десериализации.

До выхода версии Java 5 заглушки должны были создаваться вручную с использованием `rmic`. В современных версиях Java такой шаг не требуется. Тем не менее, если приходится иметь дело с очень старой унаследованной средой, то вот как с помощью `rmic` создать заглушку:

```
rmic AddServerImpl
```

Команда создает файл `AddServerImpl_Stub.class`. В случае применения `rmic` удостоверьтесь, что переменная среды `CLASSPATH` включает текущий каталог.

Шаг 3: установка файлов на клиентской и серверной машинах

Скопируйте файлы `AddClient.class`, `AddServerImpl_Stub.class` (при необходимости) и `AddServerIntf.class` в выбранный каталог на клиентской машине. Скопируйте `AddServerIntf.class`, `AddServerImpl.class`, `AddServerImpl_Stub.class` (при необходимости) и `AddServer.class` в выбранный каталог на серверной машине.

Шаг 4: запуск реестра RMI на серверной машине

В состав JDK входит программа `rmiregistry`, которая выполняется на серверной машине. Она сопоставляет имена с объектными ссылками. Первым делом нужно удостовериться в том, что переменная среды `CLASSPATH` включает каталог, в котором находятся файлы приложения, после чего запустить реестр RMI из командной строки:

```
start rmiregistry
```

Когда команда возвратит управление, на экране появится новое окно, которое необходимо оставить открытым до завершения экспериментов с примером приложения RMI.

Шаг 5: запуск сервера

Код сервера запускается из командной строки следующим образом:

```
java AddServer
```

Вспомните, что код `AddServer` создает экземпляр `AddServerImpl` и регистрирует его с именем `"AddServer"`.

Шаг 6: запуск клиента

Для программы `AddClient` требуются три аргумента: имя или IP-адрес сервера и два числа, которые нужно просуммировать. Ее можно запустить из командной строки с использованием одного из двух показанных ниже команд:

```
java AddClient server1 8 9
java AddClient 11.12.13.14 8 9
```

В первой команде указывается имя сервера, а во второй — его IP-адрес (11.12.13.14).

Опробовать это приложение можно, даже не имея удаленного сервера. Понадобится просто установить все программы на один и тот же компьютер, запустить `rmiregistry`, затем `AddServer` и выполнить `AddClient` с помощью следующей команды:

```
java AddClient 127.0.0.1 8 9
```

Здесь 127.0.0.1 — адрес обратной связи для локальной машины. Его указание позволяет задействовать весь механизм RMI без фактической установки сервера на удаленном компьютере. (Правда, при включенном брандмауэре такой подход может не сработать.)

В любом случае вот как выглядит пример вывода, генерируемого программой:

```
Первое число: 8
Второе число: 9
Сумма: 17.0
```

Форматирование даты и времени с помощью пакета java.text

Пакет java.text позволяет проводить форматирование, синтаксический разбор, поиск и манипулирование текстом. Начиная с версии JDK 9, пакет java.text является частью модуля java.base. В настоящем разделе рассматриваются два наиболее часто применяемых класса java.text, форматирующие информацию о дате и времени. Однако первым делом важно отметить, что новый API даты и времени, описанный далее в главе, предлагает современный подход к обработке даты и времени, который также поддерживает форматирование. Конечно, обсуждаемые здесь классы продолжат использоваться в унаследованном коде на протяжении некоторого времени.

Класс DateFormat

Абстрактный класс DateFormat предоставляет возможность форматирования и разбора даты и времени. Метод getDateInstance() возвращает экземпляр DateFormat, который может форматировать информацию о дате. Он доступен в следующих формах:

```
static final DateFormat getDateInstance()
static final DateFormat getDateInstance(int style)
static final DateFormat getDateInstance(int style, Locale locale)
```

В style указывается одно из значений DEFAULT, SHORT, MEDIUM, LONG или FULL, которые представляют собой константы типа int, определенные в DateFormat. Они отвечают за выдачу различной информации о дате. В locale задается локаль (детальные сведения о локалях ищите в главе 21). Если значения style и/или locale не указаны, тогда применяются стандартные значения.

Одним из наиболее часто используемых методов класса DateFormat является format(), имеющий несколько перегруженных форм, одна из которых показана ниже:

```
final String format(Date d)
```

В d передается объект Date, который необходимо отобразить. Метод возвращает строку, содержащую отформатированную информацию.

В следующем листинге демонстрируется форматирование информации о дате. Сначала создается объект Date, фиксирующий информацию о текущей дате и времени. Затем информация о дате выводится с применением разных стилей и локалей.

```
// Демонстрация форматов даты.
import java.text.*;
import java.util.*;
public class DateFormatDemo {
  public static void main(String[] args) {
    Date date = new Date();
    DateFormat df;
```

```
    df = DateFormat.getDateInstance(DateFormat.SHORT, Locale.JAPAN);
    System.out.println("Япония: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.KOREA);
    System.out.println("Корея: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.LONG, Locale.UK);
    System.out.println("Соединенное Королевство: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.FULL, Locale.US);
    System.out.println("США: " + df.format(date));
  }
}
```

Вот пример вывода, генерируемого программой:

```
Япония: 2021/06/30
Корея: 2021. 6. 30.
Соединенное Королевство: 30 June 2021
США: Wednesday, June 30, 2021
```

Метод `getTimeInstance()` возвращает экземпляр `DateFormat`, который может форматировать информацию о времени. Он доступен в следующих формах:

```
static final DateFormat getTimeInstance()
static final DateFormat getTimeInstance(int style)
static final DateFormat getTimeInstance(int style, Locale locale)
```

В `style` указывается одно из значений `DEFAULT`, `SHORT`, `MEDIUM`, `LONG` или `FULL`, которые представляют собой константы типа `int`, определенные в `DateFormat`. Они отвечают за выдачу различной информации о времени. В `locale` задается локаль. Если значения `style` и/или `locale` не указаны, тогда используются стандартные значения.

В показанном далее листинге демонстрируется форматирование информации о времени. Сначала создается объект `Date`, фиксирующий текущую информацию о дате и времени. Затем информация о времени выводится с применением различных стилей и локалей.

```
// Демонстрация форматов времени.

import java.text.*;
import java.util.*;

public class TimeFormatDemo {
  public static void main(String[] args) {
    Date date = new Date();
    DateFormat df;

    df = DateFormat.getTimeInstance(DateFormat.SHORT, Locale.JAPAN);
    System.out.println("Япония: " + df.format(date));

    df = DateFormat.getTimeInstance(DateFormat.LONG, Locale.UK);
    System.out.println("Соединенное Королевство: " + df.format(date));

    df = DateFormat.getTimeInstance(DateFormat.FULL, Locale.CANADA);
    System.out.println("Канада: " + df.format(date));
  }
}
```

Вот пример вывода, генерируемого программой:

```
Япония: 13:03
Соединенное Королевство: 13:03:31 GMT-05:00
Канада: 1:03:31 PM Central Daylight Time
```

В классе DateFormat также имеется метод getDateTimeInstance(), который способен форматировать информацию о дате и времени. При желании можете поэкспериментировать с ним самостоятельно.

Класс SimpleDateFormat

SimpleDateFormat — конкретный подкласс DateFormat, который позволяет определять собственные шаблоны форматирования, используемые для отображения информации о дате и времени.

Ниже показан один из его конструкторов:

```
SimpleDateFormat(String formatString)
```

Аргумент formatString описывает, каким образом отображается информация о дате и времени. Вот пример его применения:

```
SimpleDateFormat sdf = SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
```

Символы, используемые в строке форматирования, определяют отображаемую информацию. В табл. 31.4 приведены краткие описания таких символов.

Таблица 31.4. Символы строки форматирования для SimpleDateFormat

Символ	Описание
a	AM (до полудня) или PM (после полудня)
d	День месяца (1–31)
h	Часы в формате AM/PM (1–12)
k	Часы в сутках (1–24)
m	Минуты в часе (0–59)
s	Секунды в минуте (0–59)
u	День недели (с понедельником, соответствующим 1)
w	Неделя в году (1–52)
y	Год
z	Часовой пояс
D	День в году (1–366)
E	День недели (например, Thursday (четверг))
F	День недели в месяце
G	Эра (например, AD (н.э.) или BC (до н.э.))
H	Часы в сутках (0–23)

Окончание табл. 31.4

Символ	Описание
K	Часы в формате AM/PM (0–11)
L	Месяц
M	Месяц
S	Миллисекунды в секунде
W	Неделя в месяце (1–5)
X	Часовой пояс в формате ISO 8601
Y	Неделя в году
Z	Часовой пояс в формате RFC 822

В большинстве случаев количество повторений символа определяет способ представления даты. Текстовая информация отображается в сокращенном виде, если буква шаблона повторяется менее четырех раз. В противном случае применяется несокращенная форма. Скажем, шаблон `zzzz` позволяет отобразить `Pacific Daylight Time` (тихоокеанское летнее время), а шаблон `zzz` — `PDT`. Для чисел количество повторений буквы шаблона определяет количество представленных цифр. Например, `hh:mm:ss` может представлять `01:51:15`, но `h:m:s` отображает то же значение времени в виде `1:51:15`.

Наконец, `M` или `MM` обеспечивают отображение месяца с помощью одной или двух цифр, но три или более повторений `M` приводят к тому, что месяц отображается в виде текстовой строки.

Использование класса `SimpleDateFormat` демонстрируется в следующей программе:

```
// Демонстрация использования SimpleDateFormat.
import java.text.*;
import java.util.*;
public class SimpleDateFormatDemo {
  public static void main(String[] args) {
    Date date = new Date();
    SimpleDateFormat sdf;
    sdf = new SimpleDateFormat("hh:mm:ss");
    System.out.println(sdf.format(date));
    sdf = new SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
    System.out.println(sdf.format(date));
    sdf = new SimpleDateFormat("E MMM dd yyyy");
    System.out.println(sdf.format(date));
  }
}
```

Вот пример вывода:

```
01:30:51
30 Jun 2021 01:30:51 CDT
Wed Jun 30 2021
```

Пакеты java.time, поддерживающие API даты и времени

В главе 21 обсуждался давно устоявшийся подход Java к обработке даты и времени с применением таких классов, как Calendar и GregorianCalendar. Ожидается, что этот традиционный подход еще какое-то время будет широко использоваться и потому с ним должны быть знакомы все программисты на Java. Начиная с версии JDK 8, в Java предлагается другой подход к обработке времени и даты, который реализован в пакетах, кратко описанных в табл. 31.5.

Таблица 31.5. Пакеты java.time

Пакет	Описание
java.time	Предоставляет классы верхнего уровня, которые поддерживают время и дату
java.time.chrono	Поддерживает альтернативные календари, отличающиеся от григорианского
java.time.format	Поддерживает форматирование времени и даты
java.time.temporal	Поддерживает расширенную функциональность даты и времени
java.time.zone	Поддерживает часовые пояса

Указанные в табл. 31.5 пакеты содержат большое количество классов, интерфейсов и перечислений, которые обеспечивают обширную и детализированную поддержку операций со значениями времени и даты. По причине большого количества элементов новый API времени и даты поначалу может показаться слегка пугающим. Тем не менее, он хорошо организован и логически структурирован. Его размер отражает обеспечиваемый уровень контроля и гибкости. Хотя рассмотрение абсолютно всех элементов такого обширного API выходит далеко за рамки книги, будет рассмотрено несколько фундаментальных классов. Вы увидите, что их вполне достаточно для достижения многих целей. Начиная с версии JDK 9, пакеты java.time располагаются в модуле java.base.

Фундаментальные классы для поддержки даты и времени

В java.time определен ряд классов верхнего уровня, которые предоставляют простой доступ к дате и времени. Тремя из них являются LocalDate, LocalTime и LocalDateTime. Как следует из их имен, они инкапсулируют локальную дату, время, а также дату и время. С применением упомянутых классов легко получить текущую дату и время, отформатировать дату и время и сравнить дату и время, не считая прочие операции. Как и многие другие

классы в java.time, они основаны на значениях. (Классы, основанные на значениях, обсуждались в главе 13.)

Класс LocalDate инкапсулирует дату, которая использует стандартный григорианский календарь согласно ISO 8601. Класс LocalTime инкапсулирует время в соответствии с ISO 8601. Класс LocalDateTime инкапсулирует дату и время. Эти классы содержат большое количество методов, которые обеспечивают доступ к компонентам даты и времени, позволяют сравнивать даты и время, добавлять или вычитать компоненты даты или времени и т.д. Поскольку для методов применяется общее соглашение об именовании, научившись пользоваться одним из классов, вам будет несложно освоить и другие.

Открытые конструкторы в классах LocalDate, LocalTime и LocalDateTime не определены. Взамен для получения экземпляра будет применяться фабричный метод. Одним очень удобным методом является now(), который определен во всех трех классах и возвращает текущую дату и/или время системы. В каждом классе определено несколько его форм, но здесь будет использоваться простейшая из них. Вот версия из LocalDate:

```
static LocalDate now()
```

Ниже показана версия из LocalTime:

```
static LocalTime now()
```

А версия из LocalDateTime выглядит так:

```
static LocalDateTime now()
```

В каждом случае возвращается соответствующий объект. Объект, возвращаемый методом now(), может отображаться в своем стандартном удобочитаемом виде с помощью, например, оператора println(). Однако возможен также полный контроль над форматированием даты и времени.

В следующей программе посредством классов LocalDate и LocalTime получается текущая дата и время, после чего они отображаются. Обратите внимание на вызов метода now() для получения текущей даты и времени.

```
// Простой пример использования LocalDate и LocalTime.

import java.time.*;

class DateTimeDemo {
  public static void main(String[] args) {

    LocalDate curDate = LocalDate.now();
    System.out.println(curDate);

    LocalTime curTime = LocalTime.now();
    System.out.println(curTime);
  }
}
```

Ниже показан пример вывода:

```
2021-06-30
14:57:29.621839100
```

Вывод отражает стандартный формат, заданный для даты и времени. (В следующем разделе объясняется, как указать другой формат.)

Поскольку предыдущая программа отображает и текущую дату, и текущее время, ее было бы проще написать с применением класса `LocalDateTime`. При таком подходе потребуется создать только один экземпляр и сделать один вызов `now()`:

```
LocalDateTime curDateTime = LocalDateTime.now();
System.out.println(curDateTime);
```

В результате стандартный вывод включает дату и время, например:

```
2021-06-30T14:58:56.498907300
```

И еще один момент: из экземпляра `LocalDateTime` можно получить ссылку на компонент даты или времени с использованием методов `toLocalDate()` и `toLocalTime()`:

```
LocalDate toLocalDate()
LocalTime toLocalTime()
```

Каждый метод возвращает ссылку на указанный элемент.

Форматирование даты и времени

Хотя стандартные форматы, показанные в предыдущих примерах, подходят для некоторых целей, часто может требоваться другой формат. К счастью, сменить формат легко, т.к. классы `LocalDate`, `LocalTime` и `LocalDateTime` предлагают метод `format()`:

```
String format(DateTimeFormatter fmtr)
```

В `fmtr` указывается экземпляр `DateTimeFormatter`, который предоставит нужный формат.

Класс `DateTimeFormatter` находится в пакете `java.time.format`. Для получения экземпляра `DateTimeFormatter` обычно применяется один из его фабричных методов, три из которых показаны ниже:

```
static DateTimeFormatter ofLocalizedDate(FormatStyle fmtDate)
static DateTimeFormatter ofLocalizedTime(FormatStyle fmtTime)
static DateTimeFormatter ofLocalizedDateTime(FormatStyle fmtDate,
                                             FormatStyle fmtTime)
```

Разумеется, создаваемый экземпляр `DateTimeFormatter` будет основан на типе объекта, с которым планируется его работа. Скажем, при желании отформатировать дату в экземпляре `LocalDate` необходимо использовать `ofLocalizedDate()`. Конкретный формат задается параметром `FormatStyle`.

В перечислении `FormatStyle` из пакета `java.time.format` определены следующие константы:

```
FULL
LONG
MEDIUM
SHORT
```

С их помощью указывается уровень детализации при отображении. (Таким образом, данная форма DateTimeFormatter функционирует аналогично классу java.text.DateFormat, описанному ранее в главе.)

В приведенном далее примере демонстрируется применение класса DateTimeFormatter для отображения текущей даты и времени:

```
// Демонстрация использования DateTimeFormatter.
import java.time.*;
import java.time.format.*;
class DateTimeDemo2 {
  public static void main(String[] args) {
    LocalDate curDate = LocalDate.now();
    System.out.println(curDate.format(
          DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)));
    LocalTime curTime = LocalTime.now();
    System.out.println(curTime.format(
          DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT)));
  }
}
```

Ниже показан вывод:

```
Wednesday, June 30, 2021
2:16 PM
```

В ряде ситуаций требуется формат, отличающийся от тех, которые можно указать посредством FormatStyle. Один способ достичь такой цели предусматривает использование предопределенного форматировщика вроде ISO_DATE или ISO_TIME, предоставляемого DateTimeFormatter, а другой способ связан с созданием собственного формата за счет указания шаблона. Для этого можно применить фабричный метод ofPattern() класса DateTimeFormatter. Вот одна из его форм:

```
static DateTimeFormatter ofPattern(String fmtPattern)
```

В fmtPattern передается строка, содержащая необходимый шаблон даты и времени. Метод ofPattern() возвращает экземпляр DateTimeFormatter, который будет форматировать согласно указанному шаблону. Используется стандартная локаль.

В общем случае шаблон состоит из спецификаторов формата, называемых *буквами шаблона*. Буква шаблона будет заменена соответствующим ей компонентом даты или времени. Полный список букв шаблона приведен в документации API для метода ofPattern(), а в табл. 31.6 описаны избранные буквы. Обратите внимание, что буквы шаблона чувствительны к регистру.

Точный вывод будет определяться тем, сколько раз повторяется буква шаблона. (Таким образом, DateTimeFormatter работает примерно так же, как класс java.text.SimpleDateFormat, описанный ранее в главе.) Скажем, для месяца апреля следующие шаблоны:

```
M MM MMM MMMM
```

обеспечат такой вывод:

```
4 04 Apr April
```

Таблица 31.6. Избранные буквы шаблона

Буква	Описание
a	Указатель AM/PM
d	День месяца
E	День недели
h	Часы в 12-часовой форме
H	Часы в 24-часовой форме
M	Месяц
m	Минуты
s	Секунды
y	Год

Откровенно говоря, экспериментирование — лучший способ понять, что делает каждая буква шаблона и каким образом различные повторения влияют на результат.

Чтобы вывести букву шаблона в виде текста, ее понадобится заключить в одинарные кавычки. Как правило, рекомендуется помещать в одинарные кавычки все символы, не относящиеся к шаблону, во избежание проблем, связанных с возможным изменением набора букв шаблона в последующих версиях Java.

В показанной ниже программе демонстрируется применение шаблона даты и времени:

```
// Создание специального формата для вывода даты и времени.
import java.time.*;
import java.time.format.*;

class DateTimeDemo3 {
  public static void main(String[] args) {

    LocalDateTime curDateTime = LocalDateTime.now();
    System.out.println(curDateTime.format(
            DateTimeFormatter.ofPattern("MMMM d',' yyyy h':'mm a")));
  }
}
```

Вот пример вывода, генерируемого программой:

```
June 30, 2021 2:22 PM
```

Еще один момент, касающийся создания специального формата для вывода даты и времени: в классах `LocalDate`, `LocalTime` и `LocalDateTime` определены методы, позволяющие получать различные компоненты даты и времени. Например, метод `getHour()` возвращает часы в виде значения `int`; метод `getMonth()` возвращает месяц как значение перечисления `Month`; метод `getYear()` возвращает год в виде значения `int`. С использованием упо-

мянутых и других методов вывод можно формировать вручную. Кроме того выдаваемые ими значения можно применять для других целей, скажем, создавая специализированные таймеры.

Разбор строк с датой и временем

Классы `LocalDate`, `LocalTime` и `LocalDateTime` предоставляют возможность разбора строк с датой и/или временем, для чего нужно вызвать метод `parse()` на экземпляре одного из классов. Метод `parse()` имеет две формы. Его первая форма использует стандартный форматировщик, который выполняет разбор строки с датой и/или временем в стиле ISO, например, 03:31 для времени и 2021-08-02 для даты. Ниже показана первая форма метода `parse()` для класса `LocalDateTime`. (Формы для других классов аналогичны, не считая возвращаемого типа.)

```
static LocalDateTime parse(CharSequence dateTimeStr)
```

В `dateTimeStr` указывается строка, которая содержит дату и время в надлежащем формате. В случае недопустимого формата генерируется исключение.

Для разбора строки с датой и/или временем в формате, отличающемся от ISO, применяется вторая форма метода `parse()`, которая позволяет указать собственный форматировщик. Далее представлена вторая форма метода `parse()` для класса `LocalDateTime`. (Формы для других классов аналогичны, не считая возвращаемого типа.)

```
static LocalDateTime parse(CharSequence dateTimeStr,
                           DateTimeFormatter dateTimeFmtr)
```

В `dateTimeFmtr` указывается используемый форматировщик.

Ниже приведен простой пример, в котором выполняется разбор строки с датой и временем за счет применения специального форматировщика:

```
// Разбор строки с датой и временем.

import java.time.*;
import java.time.format.*;

class DateTimeDemo4 {
  public static void main(String[] args) {

    // Получить объект LocalDateTime путем разбора строки с датой и временем
    LocalDateTime curDateTime =
      LocalDateTime.parse("June 30, 2021 12:01 AM",
        DateTimeFormatter.ofPattern("MMMM d',' yyyy hh':'mm a"));

    // Отобразить результаты разбора.
    System.out.println(curDateTime.format(
            DateTimeFormatter.ofPattern("MMMM d',' yyyy h':'mm a")));
  }
}
```

Вот пример вывода, генерируемого программой:

```
June 30, 2021 12:01 AM
```

Дальнейшее исследование пакета java.time

Исследование всех пакетов для работы с датой и временем лучше всего начать с пакета java.time, который содержит большой объем полезной функциональности. Первым делом изучите методы, определенные в классах LocalDate, LocalTime и LocalDateTime. Каждый класс имеет методы, которые позволяют среди прочего добавлять или вычитать значения даты и/или времени, корректировать значения даты и/или времени по заданному компоненту, сравнивать значения даты и/или времени и создавать экземпляры на основе компонентов даты и/или времени. Вас могут заинтересовать и другие классы из пакета java.time, такие как Instant, Duration и Period. Класс Instant инкапсулирует момент времени, класс Duration — длительность во времени, Period — промежуток между датами. Вдобавок рекомендуется ознакомиться с новым интерфейсом InstantSource, который появился в версии JDK 17 и реализован классом Clock.

ЧАСТЬ

III

Введение в программирование графических пользовательских интерфейсов с помощью Swing Java

ГЛАВА

32 Введение в Swing

В части II книги было показано, как создавать очень простые пользовательские интерфейсы с помощью классов AWT. Хотя библиотека AWT попрежнему является важной частью Java, набор ее компонентов больше не используется широко для создания графических пользовательских интерфейсов. В настоящее время программисты для такой цели обычно применяют Swing — инфраструктуру, которая предлагает более мощные и гибкие компоненты для графических пользовательских интерфейсов по сравнению с AWT. В результате именно Swing повсеместно используется программистами на Java уже более двух десятилетий.

Инфраструктура Swing раскрывается в трех главах. В этой главе предлагается введение в Swing. Она начинается с описания основных концепций Swing. Затем рассматривается простой пример, демонстрирующий общую форму программы Swing. Далее следует пример, в котором применяется обработка событий. Глава завершается объяснением того, как осуществляется рисование в Swing. В следующей главе описано несколько часто используемых компонентов Swing. В третьей главе представлены меню на основе Swing. Важно понимать, что количество классов и интерфейсов в пакетах Swing довольно велико, и их невозможно охватить в данной книге. (На самом деле полное описание инфраструктуры Swing заняло бы отдельную книгу.) Однако предложенные три главы дадут общее представление о такой важной теме, как Swing.

На заметку! Исчерпывающее введение в Swing ищите в книге *Swing: руководство для начинающих* (пер. с англ., ИД "Вильямс", 2007 г.).

Происхождение инфраструктуры Swing

В первоначальных версиях Java инфраструктура Swing не существовала. Скорее, она стала ответом на недостатки исходной подсистемы для построения графических пользовательских интерфейсов Java: библиотеки Abstract Window Toolkit (AWT), в которой был определен базовый набор элементов управления, окон и диалоговых окон, поддерживающих удобный, но ограни-

ченный графический интерфейс. Одна из причин ограниченной природы библиотеки AWT заключалась в том, что она транслировала свои визуальные компоненты в соответствующие зависимые от платформы эквиваленты, или *равноправные компоненты.* Таким образом, внешний вид компонента определяется платформой, а не Java. Поскольку компоненты AWT применяли собственные кодовые ресурсы, они назывались *тяжеловесными.*

Использование собственных равноправных компонентов привело к возникновению нескольких проблем. Во-первых, из-за различий между операционными системами компонент мог выглядеть или даже действовать поразному на разных платформах. Такая потенциальная изменчивость угрожала всеобъемлющей философии Java: "написанное однажды выполняется везде, в любое время, всегда". Во-вторых, внешний вид каждого компонента был фиксированным (потому что он определялся платформой) и не мог достаточно легко изменяться. В-третьих, применение тяжеловесных компонентов порождало некоторые неприятные ограничения. Например, тяжеловесный компонент всегда был непрозрачным.

Вскоре после первоначального выпуска Java стало очевидным, что ограничения, присутствующие в AWT, достаточно серьезны, и необходим лучший подход. Решением выступила инфраструктура Swing, которая была представлена в 1997 году и включена в состав JFC (Java Foundation Classes — фундаментальные классы Java). Первоначально инфраструктура Swing была доступна для использования с Java 1.1 в виде отдельной библиотеки. Тем не менее, начиная с версии Java 1.2, инфраструктура Swing и остальная часть JFC были полностью интегрированы в Java.

Инфраструктура Swing построена на основе AWT

Прежде чем двигаться дальше, необходимо сделать одно важное замечание: хотя инфраструктура Swing устраняет ряд ограничений, присущих библиотеке AWT, она ее *не заменяет.* Напротив, инфраструктура Swing построена на основе AWT — вот почему AWT по-прежнему является важной частью Java. В Swing также применяется тот же механизм обработки событий, что и в AWT. Поэтому для использования Swing требуется базовое понимание AWT и обработки событий. (Библиотека AWT рассматривалась в главах 26 и 27. Обработка событий была описана в главе 25.)

Две ключевые особенности Swing

Как объяснялось выше, инфраструктура Swing была создана для устранения ограничений, присутствующих в AWT, что достигается благодаря двум ключевым особенностям: легковесным компонентам и подключаемому внешнему виду. Вместе они обеспечивают элегантное, но простое в применении решение проблем, присущих AWT. Именно указанные две особенности определяют сущность Swing. Ниже они рассматриваются по очереди.

Компоненты Swing являются легковесными

За редким исключением компоненты Swing *легковесны*, т.е. они полностью написаны на Java и не сопоставляются напрямую с равноправными компонентами, специфичными для платформы. Таким образом, легковесные компоненты обладают более высокой эффективностью и гибкостью. Кроме того, поскольку легковесные компоненты не транслируются в собственные равноправные компоненты, внешний вид каждого компонента определяется инфраструктурой Swing, а не лежащей в основе операционной системой. В итоге каждый компонент будет работать единообразно на всех платформах.

Инфраструктура Swing поддерживает подключаемый внешний вид

Инфраструктура Swing поддерживает *подключаемый внешний вид* (pluggable look and feel — PLAF). Так как каждый компонент Swing визуализируется кодом Java, а не собственными равноправными компонентами, внешний вид компонента находится под контролем Swing. Данный факт означает, что можно отделить внешний вид компонента от его логики, чем собственно и занимается Swing. Отделение внешнего вида дает значительное преимущество: появляется возможность изменить способ визуализации компонента, не затрагивая другие его аспекты. Другими словами, к любому компоненту можно "подключить" новый внешний вид, не создавая никаких побочных эффектов в коде, использующем этот компонент. Более того, несложно даже определять целые наборы внешних видов, которые представляют различные стили графического пользовательского интерфейса. Для применения определенного стиля просто "подключается" его внешний вид, после чего все компоненты автоматически визуализируются с использованием данного стиля.

Подключаемый внешний вид обеспечивает ряд важных преимуществ. Скажем, можно определить внешний вид, который будет выглядеть согласованным на всех платформах. И наоборот, можно создать внешний вид, предназначенный для специфической платформы. Также есть возможность спроектировать специальный внешний вид. Наконец, внешний вид можно динамически изменять во время выполнения.

В Java предоставляются внешние виды вроде metal и Nimbus, которые доступны всем пользователям Swing. Внешний вид metal также называют *внешним видом Java*. Он не зависит от платформы и доступен во всех исполняющих средах Java. Вдобавок он выбирается по умолчанию. В книге применяется стандартный внешний вид Java (metal), т.к. он не зависит от платформы.

Связь с архитектурой MVC

В общем случае визуальный компонент характеризуют три разных аспекта:

- внешний вид компонента при визуализации на экране;
- способ реагирования компонента на действия пользователя;
- информация о состоянии, ассоциированная с компонентом.

Независимо от того, какая архитектура используется для реализации компонента, он должен неявно содержать указанные три части. Существует архитектура компонентов, которая за прошедшие годы доказала свою исключительную эффективность: *модель-представление-контроллер* (Model-View-Controller — MVC).

Успех архитектуры MVC обусловлен тем, что каждая ее часть соответствует отдельному аспекту компонента. В рамках терминологии MVC *модель* соответствует информации о состоянии, связанной с компонентом. Например, в случае флажка модель содержит поле, которое указывает, установлен флажок или нет. *Представление* определяет, каким образом компонент отображается на экране, включая любые аспекты вида, на которые влияет текущее состояние модели. *Контроллер* устанавливает, как компонент реагирует на действия пользователя.

Скажем, когда пользователь щелкает на флажке, контроллер реагирует изменением модели, отражая выбор пользователя (отмечен или не отмечен), что приводит к обновлению представления. За счет разделения компонента на модель, представление и контроллер можно изменить конкретную реализацию любой части, не затрагивая остальные. Например, различные реализации представления могут визуализировать один и тот же компонент поразному, не влияя на модель или на контроллер.

Хотя архитектура MVC и лежащие в ее основе принципы концептуально верны, высокий уровень разделения между представлением и контроллером невыгоден для компонентов Swing. Взамен в Swing применяется модифицированная версия MVC, которая объединяет представление и контроллер в единую логическую сущность, называемую *делегатом пользовательского интерфейса*. По этой причине подход Swing называется либо архитектурой *модель-делегат* (Model-Delegate), либо архитектурой *раздельная модель* (Separable Model). Таким образом, хотя архитектура компонентов Swing основана на MVC, она не использует классическую реализацию MVC.

Подключаемый внешний вид Swing стал возможен благодаря архитектуре "модель-делегат". Поскольку представление (внешний вид) и контроллер (поведение) отделены от модели, внешний вид и поведение можно изменять, не оказывая влияние на способ применения компонента в программе. И наоборот, можно настраивать модель, не влияя на то, каким образом компонент отображается на экране или реагирует на действия пользователя.

Для поддержки архитектуры "модель-делегат" большинство компонентов Swing содержат два объекта. Первый представляет модель, а второй — делегат пользовательского интерфейса. Модели определяются интерфейсами. Скажем, модель для кнопки определяется интерфейсом `ButtonModel`. Делегаты пользовательского интерфейса — это классы, унаследованные от `ComponentUI`. Например, в качестве делегата пользовательского интерфейса для кнопки выступает `ButtonUI`. Обычно в программах не придется взаимодействовать напрямую с делегатом пользовательского интерфейса.

Компоненты и контейнеры

Графический пользовательский интерфейс Swing состоит из двух ключевых элементов: *компонентов* и *контейнеров*. Однако различие в основном концептуальное, потому что все контейнеры тоже являются компонентами. Разница между ними кроется в их предполагаемом назначении: в широком смысле компонент представляет собой независимый визуальный элемент управления, такой как кнопка или ползунок. Контейнер содержит группу компонентов. Следовательно, контейнер — это особый тип компонента, который предназначен для хранения других компонентов. Более того, чтобы компонент отображался, он должен находиться внутри контейнера. Таким образом, все графические интерфейсы Swing будут иметь как минимум один контейнер. Поскольку контейнеры относятся к компонентам, контейнер способен также содержать другие контейнеры, позволяя инфраструктуре Swing определить то, что называется *иерархией включения*, на вершине которой должен быть *контейнер верхнего уровня*.

Давайте более подробно рассмотрим компоненты и контейнеры.

Компоненты

Как правило, компоненты Swing являются производными от класса `JComponent` (за исключением четырех контейнеров верхнего уровня, описанных в следующем разделе.) Класс `JComponent` обеспечивает функциональность, общую для всех компонентов. Например, `JComponent` поддерживает подключаемый внешний вид. Класс `JComponent` унаследован от классов `Container` и `Component` библиотеки AWT, т.е. компонент Swing построен на основе компонента AWT и совместим с ним.

Все компоненты Swing представлены классами, определенными в пакете `javax.swing`. Ниже перечислены имена классов для компонентов Swing (в том числе и те, что используются в качестве контейнеров):

`JApplet` (устаревший)	`JButton`	`JCheckBox`	`JCheckBoxMenuItem`
`JColorChooser`	`JComboBox`	`JComponent`	`JDesktopPane`
`JDialog`	`JEditorPane`	`JFileChooser`	`JFormattedTextField`
`JFrame`	`JInternalFrame`	`JLabel`	`JLayer`
`JLayeredPane`	`JList`	`JMenu`	`JMenuBar`
`JMenuItem`	`JOptionPane`	`JPanel`	`JPasswordField`
`JPopupMenu`	`JProgressBar`	`JRadioButton`	`JRadioButtonMenuItem`
`JRootPane`	`JScrollBar`	`JScrollPane`	`JSeparator`
`JSlider`	`JSpinner`	`JSplitPane`	`JTabbedPane`
`JTable`	`JTextArea`	`JTextField`	`JTextPane`
`JTogglebutton`	`JToolBar`	`JToolTip`	`JTree`
`JViewport`	`JWindow`		

Обратите внимание, что имена всех классов компонентов начинаются с буквы J. Скажем, класс для метки называется JLabel, класс для кнопки — JButton, класс для полосы прокрутки — JScrollBar и т.д.

Контейнеры

В Swing определены два вида контейнеров. Первый вид — контейнеры верхнего уровня: JFrame, JApplet, JWindow и JDialog. Они не унаследованы от JComponent, но унаследованы от классов Component и Container библиотеки AWT. В отличие от других компонентов Swing, которые легковесны, контейнеры верхнего уровня тяжеловесны. В результате контейнеры верхнего уровня представляют собой особый случай в библиотеке компонентов Swing.

Как следует из названия, контейнер верхнего уровня обязан находиться на вершине иерархии включения. Контейнер верхнего уровня не содержится в каком-то другом контейнере. Более того, каждая иерархия включения должна начинаться с контейнера верхнего уровня. Чаще всего в приложениях применяется класс JFrame. В прошлом для аплетов использовался класс JApplet. Как объяснялось в главе 1, начиная с версии JDK 9, аплеты считаются нерекомендуемыми, а теперь они стали устаревшими и подлежащими удалению. В результате класс JApplet тоже устарел и подлежит удалению. Кроме того, начиная с версии JDK 11, сама поддержка аплетов была удалена.

Ко второму виду контейнеров, поддерживаемых Swing, относятся легковесные контейнеры, которые унаследованы от JComponent. Примером легковесного контейнера может служить JPanel — контейнер общего назначения. Легковесные контейнеры часто применяются для организации и управления группами связанных компонентов, поскольку такой контейнер может содержаться внутри другого контейнера. Следовательно, легковесные контейнеры наподобие JPanel можно использовать для создания подгрупп связанных элементов управления, содержащихся во внешнем контейнере.

Панели контейнеров верхнего уровня

В каждом контейнере верхнего уровня определен набор *панелей*. На вершине иерархии находится экземпляр класса JRootPane — легковесного контейнера, который предназначен для управления другими панелями. Он также помогает управлять необязательным меню. В состав корневой панели могут входить *прозрачная панель*, *панель содержимого* и *многослойная панель*.

Прозрачная панель является панелью верхнего уровня. Она находится над всеми остальными панелями, полностью их покрывает и по умолчанию представляет собой прозрачный экземпляр JPanel. Прозрачная панель позволяет, например, управлять событиями мыши, которые оказывают воздействие на весь контейнер (а не на отдельный элемент управления), или рисовать поверх любого другого компонента. В большинстве случаев применять прозрачную панель напрямую не придется, но важно помнить, что она есть.

Многослойная панель является экземпляром JLayeredPane и дает возможность размещать компоненты на заданной глубине. Значение глубины

определяет, какой компонент перекрывает другой. (Таким образом, многослойная панель позволяет указывать Z-порядок для компонента, хотя обычно поступать так вовсе не обязательно.) Многослойная панель содержит панель содержимого и дополнительное меню.

Хотя прозрачная панель и многослойные панели являются неотъемлемой частью функционирования контейнера верхнего уровня и служат важным целям, многое из того, что они обеспечивают, происходит "за кулисами". Больше всего приложение будет взаимодействовать с панелью содержимого, потому что именно туда будут добавляться визуальные компоненты. Другими словами, когда компонент вроде кнопки добавляется в контейнер верхнего уровня, то он попадает в панель содержимого. По умолчанию панель содержимого представляет собой непрозрачный экземпляр `JPanel`.

Пакеты Swing

Инфраструктура Swing — очень крупная подсистема, которая использует множество пакетов. На момент написания книги в Swing были определены перечисленные ниже пакеты:

`javax.swing`	`javax.swing.plaf.basic`	`javax.swing.text`
`javax.swing.border`	`javax.swing.plaf.metal`	`javax.swing.text.html`
`javax.swing.colorchooser`	`javax.swing.plaf.multi`	`javax.swing.text.html.parser`
`javax.swing.event`	`javax.swing.plaf.nimbus`	`javax.swing.text.rtf`
`javax.swing.filechooser`	`javax.swing.plaf.synth`	`javax.swing.tree`
`javax.swing.plaf`	`javax.swing.table`	`javax.swing.undo`

Начиная с версии JDK 9, пакеты Swing находятся в модуле `java.desktop`.

Главным пакетом является `javax.swing`, который необходимо импортировать в любую программу, где применяется Swing. Он содержит классы, реализующие основные компоненты Swing, такие как кнопки, метки и флажки.

Простое приложение Swing

Программы Swing отличаются как от консольных программ, так и от программ на основе AWT, показанных ранее в книге. Например, в них используется другой набор компонентов и другая иерархия контейнеров, нежели в программах AWT. Программы Swing также предъявляют особые требования касательно многопоточности. Лучший способ понять структуру программы Swing — исследовать ее пример. Прежде чем начать, важно отметить, что в прошлом существовало два вида программ на Java, в которых обычно применялась инфраструктура Swing. Первый вид — настольное приложение, который используется повсеместно и рассматривается здесь. Второй вид — аплет. Поскольку аплеты в настоящее время устарели и не подходят для потребления в новом коде, в книге они не обсуждаются.

Несмотря на краткость следующей программы, в ней представлен один из способов написания приложения Swing и продемонстрировано несколько ключевых средств Swing. В программе применяются два компонента Swing: `JFrame` и `JLabel`. Класс `JFrame` является контейнером верхнего уровня, который обычно используется для приложений Swing, а класс `JLabel` — компонентом Swing, создающим метку, которая представляет собой компонент, отображающий информацию. Метка — простейший компонент Swing, потому что она пассивна, т.е. не реагирует на пользовательский ввод, просто отображая выходные данные. Для хранения экземпляра JLabel в программе применяется контейнер `JFrame`. Метка отображает короткое текстовое сообщение.

```
// Простое приложение Swing.

import javax.swing.*;

  class SwingDemo {
    SwingDemo() {
      // Создать контейнер JFrame.
      JFrame jfrm = new JFrame("A Simple Swing Application");
        // Простое приложение Swing

      // Установить начальные размеры фрейма.
      jfrm.setSize(275, 100);

      // Прекратить работу, когда пользователь закрывает приложение.
      jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Создать текстовую метку.
      JLabel jlab = new JLabel("Swing means powerful GUIs.");
        // Swing позволяет строить мощные графические пользовательские
        // интерфейсы

      // Добавить метку в панель содержимого.
      jfrm.add(jlab);

      // Отобразить фрейм.
      jfrm.setVisible(true);
    }

    public static void main(String[] args) {
      // Создать фрейм в потоке диспетчеризации событий.
      SwingUtilities.invokeLater(new Runnable() {
        public void run() { new SwingDemo();
        }
      });
    }
}
```

Программы Swing компилируются и запускаются таким же способом, как другие приложения Java. Таким образом, для компиляции программы можно использовать следующую командную строку:

```
javac SwingDemo.java
```

Запускается программа так:

```
java SwingDemo
```

В результате запуска программы появляется окно, подобное показанному на рис. 32.1.

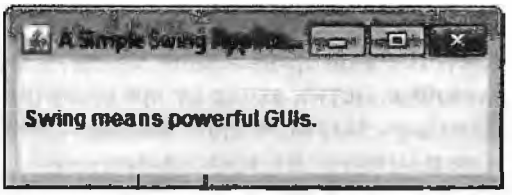


Рис. 32.1. Окно, отображаемое программой `SwingDemo`

Поскольку программа `SwingDemo` иллюстрирует несколько ключевых концепций Swing, давайте ее внимательно проанализируем, строка за строкой. Программа начинается с импорта `javax.swing`. Как уже упоминалось, пакет `javax.swing` содержит компоненты и модели, определенные инфраструктурой Swing. Например, в `javax.swing` определены классы, реализующие метки, кнопки, текстовые элементы управления и меню. Он будет включаться во все программы на базе Swing.

Затем в программе объявляется класс `SwingDemo` и конструктор для него. В конструкторе происходит большая часть действий программы. Сначала создается экземпляр `JFrame`:

```
JFrame jfrm = new JFrame("A Simple Swing Application");
```

Здесь создается контейнер по имени `jfrm`, который определяет прямоугольное окно с заголовком, кнопками закрытия, сворачивания, разворачивания и восстановления, а также системным меню. В итоге получается стандартное окно верхнего уровня. Конструктору передается заголовок окна.

Далее задаются размеры окна:

```
jfrm.setSize(275, 100);
```

Метод `setSize()`, унаследованный классом `JFrame` от класса `Component` из библиотеки AWT, устанавливает размеры окна, указанные в пикселях. Вот его общая форма:

```
void setSize(int width, int height)
```

В текущем примере ширина (`width`) окна устанавливается в 275, а высота (`height`) — в 100.

По умолчанию, когда окно верхнего уровня закрывается (например, по щелчку пользователем на кнопке закрытия), оно удаляется с экрана, но приложение не прекращает работу. Хотя в некоторых ситуациях такое стандартное поведение удобно, в большинстве приложений в нем нет необходимости. Взамен обычно нужно, чтобы при закрытии окна верхнего уровня приложение завершалось. Добиться желаемого можно несколькими способами, из которых самый простой предусматривает вызов метода `setDefaultCloseOperation()`, как сделано в программе:

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

После выполнения этого вызова закрытие окна приводит к прекращению работы всего приложения. Общая форма метода `setDefaultCloseOperation()` выглядит следующим образом:

```
void setDefaultCloseOperation(int what)
```

Передаваемое в `what` значение определяет, что происходит при закрытии окна. Помимо `JFrame.EXIT_ON_CLOSE` есть ряд других вариантов:

```
DISPOSE_ON_CLOSE
HIDE_ON_CLOSE
DO_NOTHING_ON_CLOSE
```

Их имена отражают выполняемые действия (освобождать при закрытии, скрыть при закрытии и ничего не делать при закрытии). Упомянутые константы объявлены в интерфейсе `WindowConstants` из `javax.swing`, который реализован классом `JFrame`.

Далее создается компонент `JLabel` инфраструктуры Swing:

```
JLabel jlab = new JLabel("Swing means powerful GUIs.");
```

Класс `JLabel` — самый простой и легкий в использовании компонент, т.к. он не принимает пользовательский ввод, а просто отображает информацию, которая может состоять из текста, значка либо их комбинации. Метка, созданная программой, содержит только текст, который передается ее конструктору.

В следующей строке кода метка добавляется в панель содержимого фрейма:

```
jfrm.add(jlab);
```

Как объяснялось ранее, все контейнеры верхнего уровня имеют панель содержимого, в которой хранятся компоненты. Таким образом, для помещения компонента во фрейм необходимо добавить его в панель содержимого фрейма, вызвав метод `add()` на ссылке `JFrame` (в данном случае `jfrm`). Вот общая форма метода `add()`:

```
Component add(Component comp)
```

Метод `add()` наследуется классом `JFrame` от класса `Container` из библиотеки AWT.

По умолчанию панель содержимого, связанная с `JFrame`, применяет граничную компоновку. Только что показанная форма `add()` добавляет метку в центральную область. Другие формы метода `add()` позволяют указывать одну из областей граничной компоновки. Когда компонент добавляется в центральную область, его размеры автоматически подстраиваются под размеры центральной области.

Прежде чем продолжить, нужно сделать важное замечание исторического характера. До выхода JDK 5 при добавлении компонента в панель содержимого нельзя было вызывать метод `add()` непосредственно на экземпляре `JFrame`. Взамен `add()` требовалось вызывать на панели содержимого экземпляра `JFrame`. Панель содержимого можно было получить, вызвав `getContentPane()` на экземпляре `JFrame`. Ниже показан метод `getContentPane()`:

```
Container getContentPane()
```

Он возвращает ссылку Container на панель содержимого. Затем для этой ссылки вызывался метод add(), чтобы добавить компонент в панель содержимого. Таким образом, в прошлом для добавления jlab в jfrm приходилось использовать следующий оператор:

```
jfrm.getContentPane().add(jlab); // старый стиль
```

Сначала с помощью метода getContentPane() получается ссылка на панель содержимого, после чего посредством add() компонент добавляется в контейнер, связанный с данной панелью. Аналогичная процедура требовалась также при вызове метода remove() для удаления компонента и при вызове метода setLayout() для установки диспетчера компоновки, относящегося к панели содержимого. Вот почему в унаследованном коде, написанном до появления JDK 5, часто встречаются явные вызовы getContentPane(). Теперь применять getContentPane() больше не нужно. Можно просто вызывать add(), remove() и setLayout() прямо на JFrame, потому что эти методы были изменены таким образом, что они автоматически работают с панелью содержимого.

Последний оператор в конструкторе SwingDemo делает окно видимым:

```
jfrm.setVisible(true);
```

Метод setVisible() унаследован от класса Component из AWT. Если его аргумент равен true, то окно будет отображено, а если false, тогда скрыто. По умолчанию компонент JFrame невидимый, так что для его отображения необходимо вызвать setVisible(true).

Внутри метода main() создается объект SwingDemo, который обеспечивает отображение окна и метки. Обратите внимание на то, как вызывается конструктор SwingDemo:

```
SwingUtilities.invokeLater(new Runnable() {
  public void run() {
    new SwingDemo();
  }
});
```

Приведенная кодовая последовательность создает объект SwingDemo в *потоке диспетчеризации событий*, а не в главном потоке приложения. Дело в том, что программы Swing обычно управляются событиями. Например, при взаимодействии пользователя с компонентом генерируется событие. Событие передается приложению путем вызова обработчика событий, определенного приложением. Тем не менее, обработчик выполняется в потоке диспетчеризации событий, предоставляемом Swing, а не в главном потоке приложения. Таким образом, хотя обработчики событий определены в программе, вызываются они в потоке, который программой не создавался.

Во избежание проблем (включая потенциальную взаимоблокировку) все компоненты графического пользовательского интерфейса Swing должны создаваться и обновляться из потока диспетчеризации событий, а не из главного потока приложения. Однако метод main() выполняется в главном потоке. Таким образом, main() не может напрямую создавать объект SwingDemo.

Взамен он должен создать объект Runnable, который выполняется в потоке диспетчеризации событий, и поручить этому объекту создание графического пользовательского интерфейса.

Чтобы разрешить создание кода графического пользовательского интерфейса в потоке диспетчеризации событий, необходимо воспользоваться одним из двух методов, определенных в классе SwingUtilities — invokeLater() и invokeAndWait():

```
static void invokeLater(Runnable obj)
static void invokeAndWait(Runnable obj)
  throws InterruptedException, InvocationTargetException
```

В obj передается объект Runnable, метод run() которого будет вызываться потоком диспетчеризации событий. Разница между указанными двумя методами в том, что invokeLater() возвращает значение немедленно, а invokeAndWait() ожидает возврата управления из obj.run().

Один из этих методов можно применять для вызова метода, создающего графический пользовательский интерфейс приложения Swing, или всякий раз, когда нужно изменить состояние графического пользовательского интерфейса из кода, не выполняющегося в потоке диспетчеризации событий. Обычно будет использоваться invokeLater(), как делалось в предыдущей программе. Тем не менее, при конструировании начального графического пользовательского интерфейса для аплета требуется метод invokeAndWait(). Таким образом, его часто можно встретить в унаследованном коде аплетов.

Обработка событий

В предыдущем примере была показана базовая форма программы Swing, но упущена одна важная часть: обработка событий. Поскольку компонент JLabel не принимает пользовательский ввод, он не генерирует события, поэтому обработка событий не нужна. Однако другие компоненты Swing реагируют на пользовательский ввод, и события, генерируемые такими взаимодействиями, необходимо обрабатывать. События также могут генерироваться способами, которые не связаны напрямую с пользовательским вводом. Например, событие генерируется при срабатывании таймера. В любом случае обработка событий является важной частью любого приложения на основе Swing.

В Swing применяется такой же механизм обработки событий, как и в AWT. Подход называется *моделью делегирования обработки событий* и был описан в главе 25. Во многих случаях в Swing используются те же самые события, что и в AWT, и эти события упакованы в java.awt.event. События, характерные для Swing, находятся в пакете javax.swing.event.

Несмотря на то что события обрабатываются в Swing тем же способом, как и в AWT, все-таки полезно рассмотреть простой пример. В следующей программе организуется обработка события, генерируемого кнопкой Swing. Пример вывода представлен на рис. 32.2.

```
// Обработка события в программе Swing.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class EventDemo {
  JLabel jlab;
  EventDemo() {
    // Создать контейнер JFrame.
    JFrame jfrm = new JFrame("An Event Example");
                              // Пример обработки события
    // Задать диспетчер компоновки FlowLayout.
    jfrm.setLayout(new FlowLayout());
    // Установить начальные размеры фрейма.
    jfrm.setSize(220, 90);
    // Закончить работу, когда пользователь закрывает приложение.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // Создать две кнопки.
    JButton jbtnAlpha = new JButton("Alpha");
    JButton jbtnBeta = new JButton("Beta");
    // Добавить прослушиватель событий действий для кнопки Alpha.
    jbtnAlpha.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alpha was pressed.");
                    // Нажата кнопка Alpha
      }
    });
    // Добавить прослушиватель событий действий для кнопки Beta.
    jbtnBeta.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta was pressed.");
                   // Нажата кнопка Beta
      }
    });
    // Добавить кнопки в панель содержимого.
    jfrm.add(jbtnAlpha);
    jfrm.add(jbtnBeta);
    // Создать текстовую метку.
    jlab = new JLabel("Press a button.");       // Нажмите кнопку
    // Добавить метку в панель содержимого.
    jfrm.add(jlab);
    // Отобразить фрейм.
    jfrm.setVisible(true);
  }
  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new EventDemo();
      }
    });
  }
}
```

Рис. 32.2. Вывод из программы `EventDemo`

Первым делом обратите внимание, что в программе теперь импортируются пакеты `java.awt` и `java.awt.event`. Пакет `java.awt` необходим из-за того, что он содержит класс `FlowLayout`, который поддерживает стандартный диспетчер потоковой компоновки, применяемый для размещения компонентов внутри фрейма. (Диспетчеры компоновки обсуждались в главе 27.) Пакет `java.awt.event` требуется по той причине, что в нем определен интерфейс `ActionListener` и класс `ActionEvent`.

Конструктор `EventDemo` начинается с создания компонента `JFrame` по имени `jfrm`. Затем в качестве диспетчера компоновки для панели содержимого `jfrm` устанавливается `FlowLayout`. По умолчанию панель содержимого использует диспетчер компоновки `BorderLayout`. Тем не менее, в данном примере удобнее применять `FlowLayout`.

После установки размеров и стандартной операции закрытия в `EventDemo()` создаются две кнопки:

```
JButton jbtnAlpha = new JButton("Alpha");
JButton jbtnBeta = new JButton("Beta");
```

Первая кнопка будет содержать текст `"Alpha"`, а вторая — текст `"Beta"`. Кнопки Swing являются экземплярами класса `JButton`, предоставляющего несколько конструкторов. Вот конструктор, который используется здесь:

```
JButton(String msg)
```

В `msg` указывается строка, которая будет отображаться внутри кнопки.

При нажатии кнопки генерируется событие `ActionEvent`. Таким образом, `JButton` предоставляет метод `addActionListener()`, предназначенный для добавления прослушивателя событий действий. (В классе `JButton` также есть метод `removeActionListener()`, позволяющий удалить прослушиватель, но в программе он не применяется.) Как объяснялось в главе 25, в интерфейсе `ActionListener` определен только один метод: `actionPerformed()`. Для удобства он снова показан ниже:

```
void actionPerformed(ActionEvent ae)
```

Метод `actionPerformed()` вызывается при нажатии кнопки. Другими словами, он представляет собой обработчик событий, который вызывается, когда происходит событие нажатия кнопки.

Затем с помощью следующего кода добавляются прослушиватели для событий действий кнопок:

```
// Добавить прослушиватель событий действий для кнопки Alpha.
jbtnAlpha.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("Alpha was pressed.");
  }
});
// Добавить прослушиватель событий действий для кнопки Beta.
jbtnBeta.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("Beta was pressed.");
  }
});
```

Для предоставления обработчиков событий двух кнопок используются анонимные внутренние классы. При каждом нажатии кнопки строка в метке `jlab` изменяется, чтобы отразить, какая кнопка была нажата.

Начиная с версии JDK 8, для реализации некоторых типов обработчиков событий также можно применять лямбда-выражения. Скажем, обработчик события для кнопки "Alpha" можно было бы реализовать так:

```
jbtnAlpha.addActionListener( (ae) -> jlab.setText("Alpha was pressed."));
```

Легко заметить, что код стал короче. Разумеется, выбор подхода должен быть основан на конкретной ситуации и имеющихся предпочтениях.

Далее кнопки добавляются в панель содержимого `jfrm`:

```
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);
```

Наконец, в панель содержимого добавляется метка `jlab` и окно делается видимым. После запуска программы при каждом нажатии кнопки в метке отображается сообщение, указывающее, какая кнопка была нажата.

И последнее замечание: не забывайте, что все обработчики событий, такие как `actionPerformed()`, вызываются в потоке диспетчеризации событий. Следовательно, обработчик событий должен возвращать управление быстро, чтобы не замедлять работу приложения. Если в результате возникновения события необходимо выполнить что-то, требующее много времени, то придется задействовать отдельный поток.

Рисование в Swing

Хотя набор компонентов Swing достаточно мощный, вы не ограничены только им, т.к. Swing позволяет записывать и непосредственно в область отображения фрейма, панели или какого-то другого компонента Swing, скажем, `JLabel`. Несмотря на то что во многих сценариях использования Swing рисование прямо на поверхности компонента не требуется, оно доступно для тех приложений, которым нужна такая возможность. Для записи вывода непосредственно на поверхность компонента будет применяться один или несколько методов рисования, определенных в AWT, например, `drawLine()` или `drawRect()`. Таким образом, большинство приемов и методов, описанных в главе 26, годится и для Swing. Однако существует ряд очень важных отличий, а сам процесс подробно обсуждается в настоящем разделе.

Основы рисования

Подход Swing к рисованию основан на первоначальном механизме AWT, но реализация Swing предлагает более детализированное управление. Прежде чем приступить к изучению особенностей рисования на базе Swing, полезно рассмотреть внутренний механизм, основанный на AWT.

В классе `Component` из AWT определен метод `paint()`, который предназначен для рисования прямо на поверхности компонента. Как правило, метод `paint()` в программе не вызывается. (На самом деле он должен вызываться в программе только в самых необычных случаях.) Напротив, метод `paint()` вызывается исполняющей средой всякий раз, когда компонент должен быть визуализирован. Ситуация подобного рода может возникать по нескольким причинам. Например, окно, в котором отображается компонент, может быть перекрыто другим окном, а затем снова появиться. Или же окно может быть свернуто и впоследствии восстановлено. Метод `paint()` также вызывается в начале выполнения программы. При написании кода на базе AWT метод `paint()` будет переопределяться в приложении, когда ему необходимо записывать вывод непосредственно на поверхность компонента.

Поскольку класс `JComponent` унаследован от `Component`, все легковесные компоненты Swing наследуют метод `paint()`. Тем не менее, он *не будет* переопределяться, чтобы рисовать прямо на поверхности компонента. Причина в том, что в Swing используется чуть более сложный подход к рисованию, вовлекающий три метода: `paintComponent()`, `paintBorder()` и `paintChildren()`. Они рисуют указанную часть компонента и делят процесс рисования на три отдельных логических действия. В легковесном компоненте первоначальный метод `paint()` из AWT просто вызывает эти методы в указанном выше порядке.

Для рисования на поверхности компонента Swing будет создаваться подкласс компонента и затем переопределяться его метод `paintComponent()`, который рисует внутреннюю часть компонента. Остальные два метода рисования обычно не переопределяются. При переопределении `paintComponent()` первое, что потребуется сделать — вызвать `super.paintComponent()`, чтобы выполнилась часть процесса рисования суперкласса. (Единственный случай, когда такой вызов не нужен — при полном ручном контроле над отображением компонента.) Далее понадобится записать вывод, подлежащий отображению. Вот метод `paintComponent()`:

```
protected void paintComponent(Graphics g)
```

В параметре `g` передается графический контекст, куда записывается вывод. Чтобы инициировать рисование компонента под управлением программы, необходимо вызвать метод `repaint()`, который работает в Swing точно так же, как в AWT. Метод `repaint()` определен в методе `Component`. Его вызов заставляет систему вызывать `paint()`, как только это становится возможным. Поскольку рисование является операцией, требующей много времени, такой механизм позволяет исполняющей среде мгновенно отложить рисование до

тех пор, пока, например, не будет завершена какая-то задача с более высоким приоритетом. Конечно, в инфраструктуре Swing вызов `paint()` приводит к вызову `paintComponent()`. Таким образом, для вывода на поверхность компонента программа будет хранить вывод до тех пор, пока не произойдет вызов метода `paintComponent()`. Сохраненный вывод будет рисоваться внутри переопределенного метода `paintComponent()`.

Вычисление области рисования

При рисовании на поверхности компонента важно ограничить вывод областью, находящейся внутри компонента. Хотя Swing автоматически отсекает любой вывод, выходящий за границы компонента, все же какая-то его часть может оказаться на границе и при рисовании границы будет перекрыта. Чтобы избежать такой ситуации, потребуется вычислить *область рисования* компонента, которая определяется как текущий размер компонента минус пространство, занимаемое границей. Другими словами, прежде чем рисовать компонент, нужно выяснить ширину границы и соответствующим образом подстроить рисуемый вывод.

Получить ширину границы позволяет метод `getInsets()`:

```
Insets getInsets()
```

Он определен в классе `Container` и переопределен в `JComponent`. Метод `getInsets()` возвращает объект `Insets`, содержащий размеры границы, которые можно выяснить с применением перечисленных далее полей:

```
int top;
int bottom;
int left;
int right;
```

Затем значения полей используются для вычисления области рисования с учетом ширины и высоты компонента. Получить ширину и высоту компонента можно посредством вызова методов `getWidth()` и `getHeight()` на компоненте:

```
int getWidth()
int getHeight()
```

Вычитая из ширины и высоты размеры границы, несложно вычислить полезную ширину и высоту компонента.

Пример программы рисования

Ниже показана программа, в которой демонстрируется все, что обсуждалось ранее. В ней создается класс `PaintPanel`, расширяющий `JPanel`. Объект этого класса применяется для отображения линий, конечные точки которых генерируются случайным образом. Пример вывода представлен на рис. 32.3.

```
// Рисование линий внутри панели.
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;
import java.util.*;

// Этот класс расширяет JPanel. В нем переопределяется метод
// paintComponent(), чтобы рисовать линии на поверхности панели.
class PaintPanel extends JPanel {
  Insets ins;       // хранит размеры границы панели
  Random rand;      // используется для генерации случайных чисел

  // Создать панель.
  PaintPanel() {
    // Разместить рамку вокруг панели.
    setBorder(BorderFactory.createLineBorder(Color.RED, 5));

    rand = new Random();
  }

  // Переопределить метод paintComponent().
  protected void paintComponent(Graphics g) {
    // Всегда первым вызывать метод суперкласса.
    super.paintComponent(g);

    int x, y, x2, y2;

    // Получить высоту и ширину компонента.
    int height = getHeight();
    int width = getWidth();

    // Получить размеры границы.
    ins = getInsets();

    // Нарисовать десять линий со случайным образом
    // сгенерированными конечными точками.
    for(int i=0; i < 10; i++) {
      // Получить случайные координаты, определяющие
      // конечные точки каждой линии.
      x = rand.nextInt(width-ins.left);
      y = rand.nextInt(height-ins.bottom);
      x2 = rand.nextInt(width-ins.left);
      y2 = rand.nextInt(height-ins.bottom);

      // Нарисовать линию.
      g.drawLine(x, y, x2, y2);
    }
  }
}

// Демонстрация рисования напрямую внутри панели.
class PaintDemo {

  JLabel jlab;
  PaintPanel pp;

  PaintDemo() {

    // Создать контейнер JFrame.
    JFrame jfrm = new JFrame("Paint Demo");
                          // Демонстрация рисования

    // Установить начальные размеры фрейма.
    jfrm.setSize(200, 150);
```

```
    // Закончить работу, когда пользователь закрывает приложение.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создать панель для рисования.
    pp = new PaintPanel();

    // Добавить созданную панель в панель содержимого. Из-за использования
    // стандартной граничной компоновки панель будет автоматически
    // подгоняться, чтобы уместиться в центральной области.
    jfrm.add(pp);

    // Отобразить фрейм.
    jfrm.setVisible(true);
  }

  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new PaintDemo();
      }
    });
  }
}
```

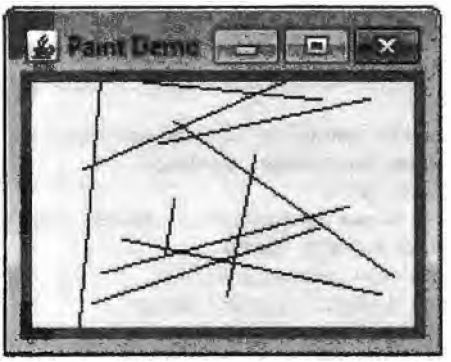


Рис. 32.3. Пример вывода из программы `PaintPanel`

Исследуем программу. Класс `PaintPanel` расширяет `JPanel` — один из легковесных контейнеров Swing, т.е. его можно добавить в панель содержимого `JFrame`. Для поддержки рисования в классе `PaintPanel` переопределяется метод `paintComponent()`, что позволит `PaintPanel` записывать прямо на поверхность компонента во время рисования. Размер панели не указан, поскольку в программе используется стандартная граничная компоновка, и панель помещается в центральную область. Это приводит к тому, что размер панели подгоняется для заполнения центральной области. В случае изменения размеров окна соответствующим образом изменятся и размеры панели.

Обратите внимание, что в конструкторе также задается рамка красного цвета шириной 5 пикселей за счет установки границы с помощью метода `setBorder()`:

```
void setBorder(Border border)
```

Интерфейс Border из Swing инкапсулирует границу. Получить границу можно вызовом одного из фабричных методов, определенных в классе BorderFactory. В программе применяется метод createLineBorder(), который создает простую линейную границу:

```
static Border createLineBorder(Color clr, int width)
```

В clr указывается цвет границы, а в width — ее ширина в пикселях. Обратите внимание, что внутри переопределенной версии метода paintComponent() сначала вызывается super.paintComponent(). Как объяснялось ранее, такой вызов необходим для обеспечения правильного рисования компонента. Далее получается ширина и высота панели вместе с размерами границы. Эти значения используются для обеспечения того, что линии находятся в пределах области рисования панели. Область рисования вычисляется путем вычитания из общей ширины и высоты компонента соответствующих размеров границы. Такие вычисления предназначены для работы с PaintPanels и границами разных размеров. Чтобы проверить, попробуйте изменить размеры окна. Все линии по-прежнему будут находиться в пределах границ панели.

Класс PaintDemo создает панель PaintPanel и добавляет ее в панель содержимого. При первом отображении окна приложения вызывается переопределенный метод paintComponent() и рисуются линии. Каждый раз, когда изменяются размеры или скрывается и восстанавливается окно, рисуется новый набор линий. Во всех случаях линии попадают в область рисования.

ГЛАВА

33 Исследование Swing

В предыдущей главе были описаны ключевые концепции, имеющие отношение к Swing, и показана общая форма приложения Swing. В настоящей главе обсуждение Swing продолжается обзором нескольких компонентов Swing, в том числе кнопок, флажков, деревьев и таблиц. Компоненты Swing обладают развитой функциональностью и допускают высокий уровень настройки. Из-за нехватки места невозможно описать все их особенности и свойства. Цель предлагаемого обзора состоит в том, чтобы дать представление о возможностях набора компонентов Swing.

Ниже перечислены классы компонентов Swing, описанные в данной главе:

`JButton`	`JCheckBox`	`JComboBox`	`JLabel`
`JList`	`JRadioButton`	`JScrollPane`	`JTabbedPane`
`JTable`	`JTextField`	`JToggleButton`	`JTree`

Все компоненты легковесны, т.е. все они являются производными от `JComponent`.

Кроме того, в главе обсуждается класс `ButtonGroup`, который инкапсулирует взаимоисключающий набор кнопок Swing, и класс `ImageIcon`, инкапсулирующий графическое изображение. Оба они определены в пакете `javax.swing` инфраструктуры Swing.

JLabel и ImageIcon

Класс `JLabel` — самый простой в использовании компонент Swing. Он создает метку и был представлен в предыдущей главе. Здесь класс `JLabel` рассматривается более подробно. Компонент `JLabel` можно применять для отображения текста и/или значка. Это пассивный компонент в том смысле, что он не реагирует на пользовательский ввод. В классе `JLabel` определено несколько конструкторов. Вот три из них:

```
JLabel(Icon icon)
JLabel(String str)
JLabel(String str, Icon icon, int align)
```

В str указывается текст, а в icon — значок для метки. В align задается выравнивание текста и/или значка по горизонтали в пределах метки с помощью константы LEFT, RIGHT, CENTER, LEADING или TRAILING. Упомянутые константы определены в интерфейсе SwingConstants вместе с рядом других констант, используемых классами Swing.

Обратите внимание, что значки указываются объектами типа Icon, который представляет собой интерфейс, определенный в Swing. Самый простой способ получения значка предусматривает применение класса ImageIcon, который реализует Icon и инкапсулирует изображение. Таким образом, объект ImageIcon можно передавать в качестве аргумента параметру типа Icon конструктора класса JLabel. Предоставить изображение можно несколькими способами, включая его чтение из файла или загрузку из URL. Ниже показан конструктор ImageIcon, который используется в примере в текущем разделе:

```
ImageIcon(String filename)
```

Он получает изображение из файла с именем, указанным в filename.

Значок и текст, ассоциированные с меткой, могут быть получены с помощью следующих методов:

```
Icon getIcon()
String getText()
```

Значок и текст, ассоциированные с меткой, могут быть установлены посредством следующих методов:

```
void setIcon(Icon icon)
void setText(String str)
```

В icon передается значок, а в str — текст. Метод setText() позволяет изменять текст внутри метки во время выполнения программы.

В следующей программе демонстрируется создание и отображение метки, содержащей значок и строку. Сначала создается объект ImageIcon для файла hourglass.png с изображением песочных часов, который применяется в качестве второго аргумента конструктора JLabel. Первым и последним аргументами конструктора JLabel являются текст метки и выравнивание. В конце метка добавляется в панель содержимого.

```
// Демонстрация использования JLabel.

import java.awt.*;
import javax.swing.*;

public class JLabelDemo {
  public JLabelDemo() {
    // Настроить JFrame.
    JFrame jfrm = new JFrame("JLabelDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(260, 210);

    // Создать значок.
    ImageIcon ii = new ImageIcon("hourglass.png");
```

```
    // Создать метку.
    JLabel jl = new JLabel("Hourglass", ii, JLabel.CENTER);

    // Добавить метку в панель содержимого.
    jfrm.add(jl);

    // Отобразить фрейм.
    jfrm.setVisible(true);
  }
  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(
      new Runnable() {
          public void run() {
            new JLabelDemo();
          }
      }
    );
  }
}
```

Вывод программы `JLabelDemo` показан на рис. 33.1.

Рис. 33.1. Вывод программы `JLabelDemo`

JTextField

Класс `JTextField` — это простейший и, вероятно, наиболее широко используемый текстовый компонент Swing. Он позволяет редактировать одну строку текста. Класс `JTextField` является производным от `JTextComponent`, который обеспечивает базовую функциональность, общую для текстовых компонентов Swing. В качестве своей модели `JTextField` применяет интерфейс `Document`. Вот три конструктора `JTextField`:

```
JTextField(int cols)
JTextField(String str, int cols)
JTextField(String str)
```

В `str` указывается строка, которая должна быть первоначально отображена, а в `cols` — количество колонок в текстовом поле. Если строка не задана, то текстовое поле изначально будет пустым. Если количество колонок не указано, тогда размер текстового поля будет подогнан к указанной строке.

Компонент JTextField генерирует события в ответ на действия пользователя. Например, событие ActionEvent инициируется, когда пользователь нажимает клавишу <Enter>, а событие CaretEvent — каждый раз, когда изменяется позиция каретки (т.е. курсора). (Класс CaretEvent расположен в пакете javax.swing.event.) Возможны и другие события. Во многих случаях обрабатывать такие события в программе не придется. Взамен при необходимости будет просто получаться строка, которая в текущий момент находится в текстовом поле. Для получения текста из текстового поля нужно вызвать метод getText().

Использование компонента JTextField демонстрируется в показанной ниже программе, где он создается и добавляется в панель содержимого. Когда пользователь нажимает клавишу <Enter>, генерируется событие действия, которое обрабатывается путем отображения текста в метке.

```
// Демонстрация использования JTextField.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JTextFieldDemo {
  public JTextFieldDemo() {
    // Настроить JFrame.
    JFrame jfrm = new JFrame("JTextFieldDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(260, 120);
    // Добавить текстовое поле в панель содержимого.
    JTextField jtf = new JTextField(15);
    jfrm.add(jtf);
    // Добавить метку.
    JLabel jlab = new JLabel();
    jfrm.add(jlab);
    // Обработать события действий.
    jtf.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        // Отобразить текст, когда пользователь нажимает ENTER.
        jlab.setText(jtf.getText());
      }
    });
    // Отобразить фрейм.
    jfrm.setVisible(true);
  }
  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JTextFieldDemo();
        }
      }
    );
  }
}
```

Вывод программы JTextFieldDemo представлен на рис. 33.2.

Рис. 33.2. Вывод программы JTextFieldDemo

Кнопки Swing

В Swing определены четыре типа кнопок: JButton, JToggleButton, JCheckBox и JRadioButton. Все они являются подклассами класса AbstractButton, который расширяет JComponent. Таким образом, все кнопки разделяют набор общих черт.

Класс AbstractButton содержит множество методов, позволяющих управлять поведением кнопок. Например, можно задавать разные значки, отображаемые для кнопки, когда она отключена, нажата или выбрана. Еще один значок можно применять в качестве *трансформируемого* значка, который отображается, когда курсор мыши располагается над кнопкой. Такие значки устанавливаются с использованием следующих методов:

```
void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
void setRolloverIcon(Icon ri)
```

Здесь di, pi, si и ri — это значки, которые должны применяться для целей, указанных методами.

Ассоциированный с кнопкой текст можно читать и устанавливать с помощью двух методов:

```
String getText()
void setText(String str)
```

В str находится текст, ассоциированный с кнопкой.

Модель, используемая всеми кнопками, определяется интерфейсом ButtonModel. При нажатии кнопка генерирует событие действия. Возможны и другие события. Далее будут рассматриваться конкретные классы кнопок.

JButton

Класс JButton обеспечивает функциональность кнопки. Его простая форма уже встречалась в предыдущей главе. Он позволяет ассоциировать с кнопкой значок, строку либо то и другое. Ниже показаны три конструктора класса JButton:

```
JButton(Icon icon)
JButton(String str)
JButton(String str, Icon icon)
```

В `str` указывается строка, а в `icon` — значок для кнопки.

При нажатии кнопки генерируется событие `ActionEvent`. С применением объекта `ActionEvent`, переданного методу `actionPerformed()` зарегистрированного прослушивателя `ActionListener`, можно получить строку с *командой действия*, ассоциированную с кнопкой. По умолчанию это строка, отображаемая внутри кнопки. Однако команду действия можно устанавливать, вызывая метод `setActionCommand()` на кнопке. Для получения команды действия понадобится вызвать метод `getActionCommand()` на объекте события:

```
String getActionCommand()
```

Команда действия идентифицирует кнопку. Таким образом, при использовании двух или более кнопок в одном приложении команда действия позволяет легко определить, какая кнопка была нажата.

В предыдущей главе приводился пример текстовой кнопки, а в следующей программе демонстрируется кнопка на основе значка. Программа отображает четыре кнопки и метку. Каждая кнопка отображает значок, представляющий часы определенного вида. При нажатии кнопки название часов отображается внутри метки.

```
// Демонстрация использования кнопки JButton на основе значка.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JButtonDemo implements ActionListener {
  JLabel jlab;

  public JButtonDemo() {

    // Настроить JFrame.
    JFrame jfrm = new JFrame("JButtonDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(500, 450);

    // Добавить кнопки в панель содержимого.
    ImageIcon hourglass = new ImageIcon("hourglass.png");
    JButton jb = new JButton(hourglass);
    jb.setActionCommand("Hourglass");          // песочные часы
    jb.addActionListener(this);
    jfrm.add(jb);

    ImageIcon analog = new ImageIcon("analog.png");
    jb = new JButton(analog);
    jb.setActionCommand("Analog Clock");       // стрелочные часы
    jb.addActionListener(this);
    jfrm.add(jb);

    ImageIcon digital = new ImageIcon("digital.png");
    jb = new JButton(digital);
    jb.setActionCommand("Digital Clock");      // цифровые часы
```

```
    jb.addActionListener(this);
    jfrm.add(jb);

    ImageIcon stopwatch = new ImageIcon("stopwatch.png");
    jb = new JButton(stopwatch);
    jb.setActionCommand("Stopwatch");      // секундомер
    jb.addActionListener(this);
    jfrm.add(jb);

    // Создать метку и добавить ее в панель содержимого.
    jlab = new JLabel("Choose a Timepiece");
                    // Выберите часы
    jfrm.add(jlab);

    // Отобразить фрейм.
    jfrm.setVisible(true);
  }

  // Обработать события кнопок.
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
                // Вы выбрали
  }

  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JButtonDemo();
        }
      }
    );
  }
}
```

Вывод программы JButtonDemo показан на рис. 33.3.

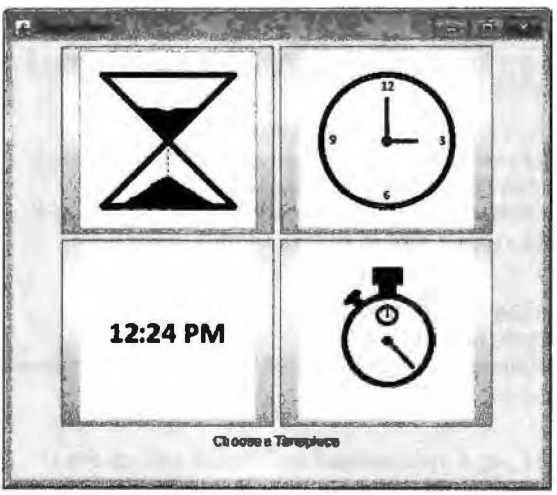


Рис. 33.3. Вывод программы JButtonDemo

JToggleButton

Полезной разновидностью кнопки является *переключатель*, который выглядит подобно кнопке, но действует по-другому, поскольку имеет два состояния: нажато и отпущено. После нажатия переключатель остается нажатым, а не отпускается, как делает обычная кнопка. После повторного нажатия переключатель отпускается. Таким образом, каждый раз, когда нажимается переключатель, он переключается между двумя состояниями.

Переключатели — это объекты класса `JToggleButton`, реализующего `AbstractButton`. Помимо создания стандартных переключателей класс `JToggleButton` является суперклассом для двух других компонентов Swing, которые тоже представляют элементы управления с двумя состояниями. Речь идет о `JCheckBox` и `JRadioButton`, рассматриваемых позже в главе. Таким образом, в `JToggleButton` определена базовая функциональность для всех компонентов с двумя состояниями.

В `JToggleButton` определено несколько конструкторов, один из которых применяется в приведенном ниже примере:

```
JToggleButton(String str)
```

Конструктор создает переключатель с текстом, переданным в `str`. По умолчанию переключатель находится в выключенном состоянии. Другие конструкторы позволяют создавать переключатели, содержащие изображения либо изображения и текст.

В классе `JToggleButton` используется модель, определенная вложенным классом по имени `JToggleButton.ToggleButtonModel`. Обычно для работы со стандартным переключателем напрямую взаимодействовать с моделью не придется.

Как и `JButton`, компонент `JToggleButton` при каждом нажатии генерирует событие действия. Тем не менее, в отличие от `JButton` компонент `JToggleButton` также генерирует событие элемента, потребляемое теми компонентами, которые поддерживают концепцию выбора. Когда переключатель `JToggleButton` нажат, он выбирается, а когда отпущен — выбор отменяется.

Для обработки событий элементов необходимо реализовать интерфейс `ItemListener`. Вспомните из главы 25, что каждый раз, когда генерируется событие элемента, оно передается методу `itemStateChanged()`, определенному в `ItemListener`. Внутри `itemStateChanged()` можно вызвать метод `getItem()` на объекте `ItemEvent`, чтобы получить ссылку на экземпляр `JToggleButton`, сгенерировавший событие:

```
Object getItem()
```

Метод возвращает ссылку на кнопку, которую придется привести к `JToggleButton`.

Самый простой способ определить состояние переключателя — вызвать метод `isSelected()` (унаследованный из `AbstractButton`) на кнопке, сгенерировавшей событие:

```
boolean isSelected()
```

Он возвращает true, если кнопка выбрана, или false в противном случае.

Ниже показан пример применения переключателя. Обратите внимание на то, как работает прослушиватель событий элементов. Он просто вызывает метод isSelected() для определения состояния кнопки.

```
// Демонстрация использования JToggleButton.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JToggleButtonDemo {
  public JToggleButtonDemo() {

    // Настроить JFrame.
    JFrame jfrm = new JFrame("JToggleButtonDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(200, 100);

    // Создать метку.
    JLabel jlab = new JLabel("Button is off.");      // Кнопка выключена

    // Создать переключатель.
    JToggleButton jtbn = new JToggleButton("On/Off");

    // Добавить прослушиватель событий элементов для переключателя.
    jtbn.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent ie) {
        if(jtbn.isSelected())
          jlab.setText("Button is on.");             // Кнопка включена
        else
          jlab.setText("Button is off.");            // Кнопка выключена
      }
    });

    // Добавить переключатель и метку в панель содержимого.
    jfrm.add(jtbn);
    jfrm.add(jlab);

    // Отобразить фрейм.
    jfrm.setVisible(true);
  }

  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JToggleButtonDemo();
        }
      }
    );
  }
}
```

Вывод программы JToggleButtonDemo представлен на рис. 33.4.

Рис. 33.4. Вывод программы `JToggleButtonDemo`

Флажки

Класс `JCheckBox` предоставляет функциональность флажка. Его непосредственным суперклассом является `JToggleButton`, который обеспечивает поддержку кнопок с двумя состояниями, как только что было описано. В классе `JCheckBox` определено несколько конструкторов, из которых здесь используется следующий:

```
JCheckBox(String str)
```

Он создает флажок, который имеет в качестве метки текст, указанный в `str`. Другие конструкторы позволяют задавать начальное состояние выбора и значок.

Когда пользователь отмечает или снимает отметку с флажка, генерируется событие `ItemEvent`. Чтобы получить ссылку на компонент `JCheckBox`, сгенерировавший событие, необходимо вызвать метод `getItem()` на объекте `ItemEvent`, который передается методу `itemStateChanged()`, определенному в `ItemListener`. Определить выбранное состояние флажка проще всего, вызвав метод `isSelected()` на экземпляре `JCheckBox`.

В следующем примере демонстрируется применение флажков. В нем отображаются четыре флажка и метка. Когда пользователь устанавливает флажок, генерируется событие `ItemEvent`. Внутри метода `itemStateChanged()` вызывается `getItem()` для получения ссылки на объект `JCheckBox`, сгенерировавший событие. Затем с помощью вызова `isSelected()` определяется, был ли флажок отмечен или же отметка была снята. Метод `getText()` получает текст для данного флажка и использует его для установки текста в метке.

```
// Демонстрация использования JCheckbox.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JCheckBoxDemo implements ItemListener {
  JLabel jlab;

  public JCheckBoxDemo() {

    // Настроить JFrame.
    JFrame jfrm = new JFrame("JCheckBoxDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(250, 100);
```

```
    // Добавить флажки в панель содержимого.
    JCheckBox cb = new JCheckBox("C");
    cb.addItemListener(this);
    jfrm.add(cb);

    cb = new JCheckBox("C++");
    cb.addItemListener(this);
    jfrm.add(cb);

    cb = new JCheckBox("Java");
    cb.addItemListener(this);
    jfrm.add(cb);

    cb = new JCheckBox("Perl");
    cb.addItemListener(this);
    jfrm.add(cb);

    // Создать метку и добавить ее в панель содержимого.
    jlab = new JLabel("Select languages");
                    // Выберите языки
    jfrm.add(jlab);

    // Отобразить фрейм.
    jfrm.setVisible(true);
  }

  // Обработать события элементов для флажков.
  public void itemStateChanged(ItemEvent ie) {
    JCheckBox cb = (JCheckBox)ie.getItem();

    if(cb.isSelected())
      jlab.setText(cb.getText() + " is selected");  // отмечен
    else
      jlab.setText(cb.getText() + " is cleared");   // не отмечен
  }
  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JCheckBoxDemo();
        }
      }
    );
  }
}
```

Вывод программы `JCheckBoxDemo` представлен на рис. 33.5.

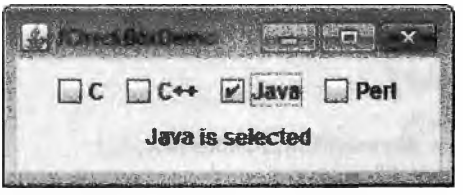


Рис. 33.5. Вывод программы `JCheckBoxDemo`

Взаимоисключающие переключатели

В группе взаимоисключающих переключателей одновременно может быть выбрана только одна кнопка. Они поддерживаются классом JRadioButton, который расширяет класс JToggleButton. В классе JRadioButton определено несколько конструкторов и вот тот, что используется в примере:

```
JRadioButton(String str)
```

В str указывается метка для кнопки. Другие конструкторы позволяют задавать начальное состояние выбора кнопки и значок.

Для активизации взаимоисключающей природы переключатели должны быть объединены в группу, где в любой момент времени можно выбрать только одну кнопку. Например, если пользователь нажимает переключатель, который находится в группе, то любой ранее выбранный переключатель в этой группе автоматически перестает быть выбранным. Группа переключателей создается классом ButtonGroup за счет вызова его стандартного конструктора. Затем элементы добавляются в группу с помощью следующего метода:

```
void add(AbstractButton ab)
```

В ab передается ссылка на кнопку, которая должна быть добавлена в группу.

Компонент JRadioButton генерирует события действий, события элементов и события изменений каждый раз, когда изменяется выбор кнопки. Чаще всего обрабатывается событие действия, т.е. обычно будет реализовываться интерфейс ActionListener. Вспомните, что в ActionListener определен единственный метод actionPerformed(). Внутри него доступно несколько способов для выяснения, какая кнопка была выбрана. Во-первых, можно проверить его команду действия, вызвав getActionCommand(). По умолчанию команда действия совпадает с меткой кнопки, но посредством вызова setActionCommand() на переключателе легко установить другую команду действия. Во-вторых, можно вызвать getSource() на объекте ActionEvent и сравнить полученную ссылку с кнопками. В-третьих, можно проверить каждый переключатель, чтобы узнать, какой из них выбран в данный момент, вызвав isSelected() на каждой кнопке. Наконец, в каждой кнопке может применяться собственный обработчик событий действий, реализованный либо в виде анонимного внутреннего класса, либо в виде лямбда-выражения. Не забывайте, что каждый раз, когда происходит событие действия, это означает, что выбранная кнопка изменилась и будет выбрана только одна кнопка.

В показанном ниже примере демонстрируется использование взаимоисключающих переключателей. Сначала создаются три переключателя, после чего они добавляются в группу. Как объяснялось ранее, это необходимо, чтобы активизировать их взаимоисключающее поведение. Нажатие переключателя генерирует событие действия, которое обрабатывается actionPerformed(). В данном обработчике с помощью метода getActionCommand() получается ассоциированный с переключателем текст, который применяется для установки текста внутри метки.

```
// Демонстрация использования JRadioButton.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JRadioButtonDemo implements ActionListener {
  JLabel jlab;

  public JRadioButtonDemo() {

    // Настроить JFrame.
    JFrame jfrm = new JFrame("JRadioButtonDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(250, 100);

    // Создать переключатели и добавить их в панель содержимого.
    JRadioButton b1 = new JRadioButton("A");
    b1.addActionListener(this);
    jfrm.add(b1);

    JRadioButton b2 = new JRadioButton("B");
    b2.addActionListener(this);
    jfrm.add(b2);

    JRadioButton b3 = new JRadioButton("C");
    b3.addActionListener(this);
    jfrm.add(b3);

    // Определить группу переключателей.
    ButtonGroup bg = new ButtonGroup();
    bg.add(b1);
    bg.add(b2);
    bg.add(b3);

    // Создать метку и добавить ее в панель содержимого.
    jlab = new JLabel("Select One");
    jfrm.add(jlab);

    // Отобразить фрейм.
    jfrm.setVisible(true);
  }

  // Обработать выбор кнопок.
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
  }

  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JRadioButtonDemo();
        }
      }
    );
  }
}
```

Вывод программы JRadioButtonDemo показан на рис. 33.6.

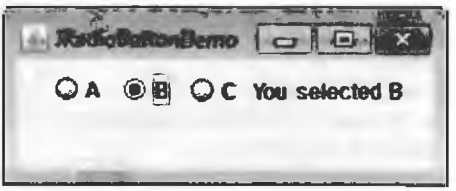


Рис. 33.6. Вывод программы JRadioButtonDemo

JTabbedPane

Класс JTabbedPane инкапсулирует *панель с вкладками*. Он управляет набором компонентов, связывая их посредством вкладок. Выбор вкладки приводит к тому, что ассоциированный с ней компонент перемещается на передний план. Панели с вкладками весьма распространены в современных графических пользовательских интерфейсах, и вы наверняка встречались с ними неоднократно. Учитывая сложную природу панели с вкладками, ее на удивление легко создавать и использовать.

В классе JTabbedPane определены три конструктора. Здесь будет применяться его стандартный конструктор, который создает пустой элемент управления с вкладками, расположенными в верхней части панели. Два других конструктора позволяют указывать расположение вкладок, которые могут располагаться вдоль любой из четырех сторон. Класс JTabbedPane использует модель SingleSelectionModel.

Вкладки добавляются вызовом метода addTab(); вот одна из его форм:

```
void addTab(String name, Component comp)
```

В name указывается имя вкладки, а в comp — компонент, который нужно добавить на вкладку. Часто компонент, добавляемый на вкладку, представляет собой панель JPanel, содержащую группу связанных компонентов. Метод addTab() позволяет вкладке содержать набор компонентов.

Общая процедура применения панели с вкладками выглядит следующим образом.

1. Создать экземпляр JTabbedPane.
2. Добавить каждую вкладку, вызывая addTab().
3. Добавить панель с вкладками в панель содержимого.

В приведенном далее примере демонстрируется использование панели с вкладками. Первая вкладка называется "Cities" (Города) и содержит четыре кнопки. На кнопках отображаются названия городов. Вторая вкладка называется "Colors" (Цвета) и содержит три флажка. В каждом флажке отображается название цвета. Третья вкладка называется "Flavors" (Ароматы) и содержит комбинированный список, который позволяет выбрать один из трех ароматов.

```
// Демонстрация использования JTabbedPane.
import javax.swing.*;
import java.awt.*;
public class JTabbedPaneDemo {
  public JTabbedPaneDemo() {
    // Настроить JFrame.
    JFrame jfrm = new JFrame("JTabbedPaneDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(400, 200);
    // Создать панель с вкладками.
    JTabbedPane jtp = new JTabbedPane();
    jtp.addTab("Cities", new CitiesPanel());
    jtp.addTab("Colors", new ColorsPanel());
    jtp.addTab("Flavors", new FlavorsPanel());
    jfrm.add(jtp);
    // Отобразить фрейм.
    jfrm.setVisible(true);
  }
  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JTabbedPaneDemo();
        }
      }
    );
  }
}
// Создать панели, которые будут добавлены в панель с вкладками.
class CitiesPanel extends JPanel {
  public CitiesPanel() {
    JButton b1 = new JButton("New York");
    add(b1);
    JButton b2 = new JButton("London");
    add(b2);
    JButton b3 = new JButton("Hong Kong");
    add(b3);
    JButton b4 = new JButton("Tokyo");
    add(b4);
  }
}
class ColorsPanel extends JPanel {
  public ColorsPanel() {
    JCheckBox cb1 = new JCheckBox("Red");
    add(cb1);
    JCheckBox cb2 = new JCheckBox("Green");
    add(cb2);
    JCheckBox cb3 = new JCheckBox("Blue");
    add(cb3);
  }
}
```

```
class FlavorsPanel extends JPanel {
  public FlavorsPanel() {
    JComboBox<String> jcb = new JComboBox<String>();
    jcb.addItem("Vanilla");
    jcb.addItem("Chocolate");
    jcb.addItem("Strawberry");
    add(jcb);
  }
}
```

Вывод программы JTabbedPaneDemo представлен на рис. 33.7.

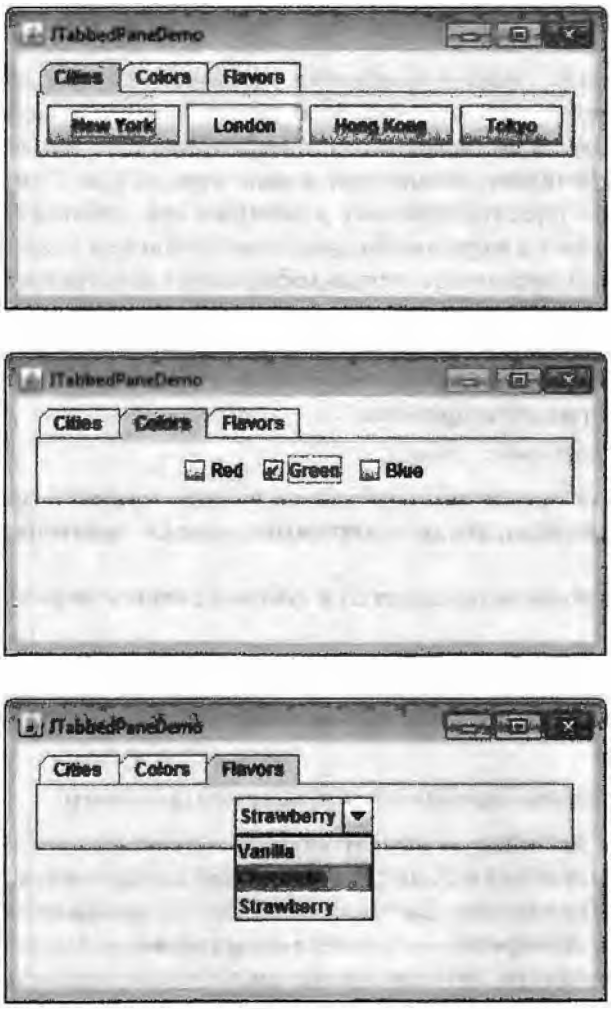


Рис. 33.7. Вывод программы JTabbedPaneDemo

JScrollPane

Класс JScrollPane представляет собой легковесный контейнер, который автоматически поддерживает прокрутку другого компонента. Прокручиваемый компонент может быть либо отдельным компонентом, скажем, таблицей, либо группой компонентов, содержащихся в другом легковесном контейнере, таком как JPanel. В любом случае, если прокручиваемый объект больше видимой области, то автоматически предоставляются горизонтальные и/или вертикальные линейки прокрутки, и компонент можно прокручивать внутри панели. Поскольку JScrollPane автоматизирует прокрутку, обычно отсутствует необходимость в управлении отдельными линейками прокрутки.

Видимая область панели прокрутки называется *окном просмотра* — там, где отображается прокручиваемый компонент. Таким образом, в окне просмотра воспроизводится видимая часть прокручиваемого компонента. Линейки прокрутки прокручивают компонент в окне просмотра. Стандартное поведение JScrollPane предусматривает динамическое добавление или удаление линейки прокрутки по мере необходимости. Например, если компонент больше по высоте окна просмотра, тогда добавляется вертикальная линейка прокрутки. Если компонент полностью умещается в окне просмотра, то линейки прокрутки удаляются.

В классе JScrollPane определено несколько конструкторов, один из которых применяется далее в примере:

```
JScrollPane(Component comp)
```

Компонент для прокрутки передается в comp. Линейки прокрутки отображаются автоматически, когда содержимое панели превышает размеры окна просмотра.

Панель прокрутки используется в соответствии с перечисленными ниже шагами.

1. Создать компонент для прокрутки.
2. Создать экземпляр JScrollPane, передавая конструктору объект для прокрутки.
3. Добавить панель прокрутки в панель содержимого.

В следующем примере демонстрируется применение области прокрутки. Сначала создается объект JPanel, к которому добавляются 400 кнопок, расположенных в 20 колонках. Затем панель JPanel добавляется в панель прокрутки, а панель прокрутки — в панель содержимого. Из-за того, что панель больше окна просмотра, автоматически появляются вертикальные и горизонтальные линейки прокрутки, которые можно использовать для перемещения по кнопкам в окне просмотра.

```
// Демонстрация использования области прокрутки.

import java.awt.*;
import javax.swing.*;
```

```
public class JScrollPaneDemo {
  public JScrollPaneDemo() {
    // Настроить JFrame. Использовать стандартную компоновку BorderLayot.
    JFrame jfrm = new JFrame("JScrollPaneDemo");
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(400, 400);

    // Создать панель и добавить к ней 400 кнопок.
    JPanel jp = new JPanel();
    jp.setLayout(new GridLayout(20, 20));
    int b = 0;
    for(int i = 0; i < 20; i++) {
      for(int j = 0; j < 20; j++) {
        jp.add(new JButton("Button " + b));
        ++b;
      }
    }

    // Создать панель прокрутки.
    JScrollPane jsp = new JScrollPane(jp);

    // Добавить панель прокрутки в панель содержимого.
    // Из-за применения стандартной компоновки панель
    // прокрутки добавится в центральную область.
    jfrm.add(jsp, BorderLayout.CENTER);

    // Отобразить фрейм.
    jfrm.setVisible(true);
  }
  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JScrollPaneDemo();
        }
      }
    );
  }
}
```

Вывод программы `JScrollPaneDemo` показан на рис. 33.8.

JList

Базовый класс списков в Swing называется `JList`. Он поддерживает выбор одного или нескольких элементов из списка. Хотя список часто состоит из строк, можно создавать списки практически из любых объектов, допускающих отображение. Класс `JList` настолько широко применяется в Java, что почти наверняка вы встречались с ним неоднократно.

В прошлом элементы в `JList` представлялись как ссылки `Object`. Однако, начиная с версии JDK 7, класс `JList` стал обобщенным и теперь он объявлен следующим образом:

```
class JList<E>
```

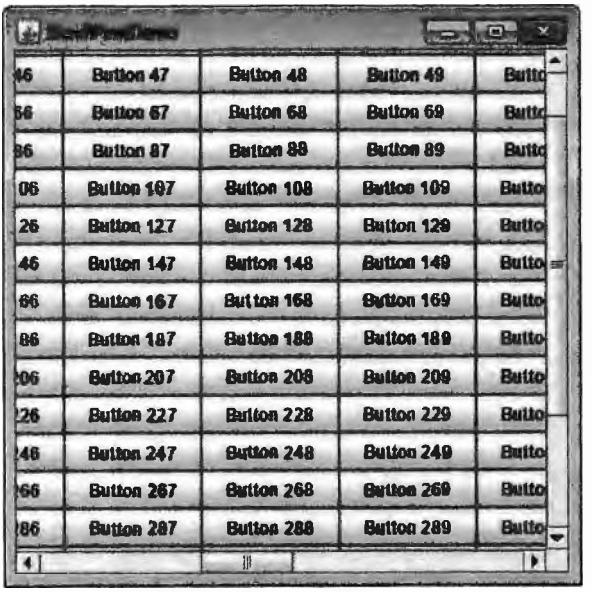


Рис. 33.8. Вывод программы `JScrollPaneDemo`

Здесь `E` представляет тип элементов в списке.

В `JList` определено несколько конструкторов, один их которых будет использоваться в главе:

```
JList(E[] items)
```

Он создает объект `JList`, содержащий элементы из массива `items`.

Класс `JList` основан на двух моделях. Первая из них — интерфейс `ListModel`, который определяет, как осуществляется доступ к данным списка. Вторая модель — интерфейс `ListSelectionModel`, предлагающий методы, которые определяют, какой элемент или элементы списка выбраны.

Несмотря на то что компонент `JList` работает надлежащим образом и сам по себе, бо́льшую часть времени он помещается внутрь `JScrollPane`. В результате длинные списки будут автоматически прокручиваться, что упрощает проектирование графических пользовательских интерфейсов и также облегчает изменение количества записей в списке без изменения размера компонента `JList`. Компонент `JList` генерирует событие `ListSelectionEvent`, когда пользователь делает выбор либо изменяет его. То же самое событие генерируется при отмене пользователем выбора элемента. Оно обрабатывается реализацией `ListSelectionListener`. В этом прослушивателе имеется только один метод по имени `valueChanged()`:

```
void valueChanged(ListSelectionEvent le)
```

В `le` передается ссылка на событие. Хотя `ListSelectionEvent` предоставляет ряд собственных методов, обычно для выяснения, что произошло, будет опрашиваться сам объект `JList`. Класс `ListSelectionEvent` и интерфейс `ListSelectionListener` находятся в пакете `javax.swing.event`.

По умолчанию компонент JList позволяет выбирать несколько диапазонов элементов в списке, но такое поведение можно изменить, вызвав метод setSelectionMode(), который определен в JList:

```
void setSelectionMode(int mode)
```

В mode указывается режим выбора, который должен быть одним из следующих значений, определенных в ListSelectionModel:

```
SINGLE_SELECTION
SINGLE_INTERVAL_SELECTION
MULTIPLE_INTERVAL_SELECTION
```

Принимаемый по умолчанию вариант MULTIPLE_INTERVAL_SELECTION дает возможность выбирать несколько интервалов элементов в списке. В случае SINGLE_INTERVAL_SELECTION можно выбирать один интервал элементов. Вариант SINGLE_SELECTION позволяет выбирать только одиночный элемент. Разумеется, одиночный элемент можно выбирать и в двух других режимах. Просто они также допускают выбор диапазона.

Чтобы получить индекс первого выбранного элемента, который также будет индексом единственного выбранного элемента в режиме SINGLE_SELECTION, понадобится вызвать метод getSelectedIndex():

```
int getSelectedIndex()
```

Индексация начинается с нуля. Таким образом, если выбран первый элемент, то getSelectedIndex() возвратит 0, а если элемент не выбран, то -1.

Вместо индекса выбранного элемента можно получить ассоциированное с ним значение, вызвав метод getSelectedValue():

```
E getSelectedValue()
```

Он возвращает ссылку на первое выбранное значение или null, если ни одно значение не выбиралось.

В следующей программе демонстрируется применение простого компонента JList, содержащего список городов. Каждый раз, когда в списке выбирается какой-нибудь город, генерируется событие ListSelectionEvent, которое обрабатывается методом valueChanged(), определенным в ListSelectionListener. Он получает индекс выбранного элемента и отображает название выбранного города внутри метки.

```
// Демонстрация использования JList.

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class JListDemo {

  // Создать список городов.
  String[] cities = { "New York", "Chicago", "Houston",
                      "Denver", "Los Angeles", "Seattle",
                      "London", "Paris", "New Delhi",
                      "Hong Kong", "Tokyo", "Sydney" };
```

```
  public JListDemo() {
    // Настроить JFrame.
    JFrame jfrm = new JFrame("JListDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(200, 200);

    // Создать JList.
    JList<String> jlst = new JList<String>(cities);

    // Установить режим выбора списка в одиночный выбор.
    jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    // Добавить список в прокручиваемую панель.
    JScrollPane jscrlp = new JScrollPane(jlst);

    // Установить предпочитаемые размеры прокручиваемой панели.
    jscrlp.setPreferredSize(new Dimension(120, 90));

    // Создать метку, которая будет отображать выбор.
    JLabel jlab = new JLabel("Choose a City"); // Выберите город

    // Добавить прослушиватель событий выбора для списка.
    jlst.addListSelectionListener(new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent le) {
        // Получить индекс измененного элемента.
        int idx = jlst.getSelectedIndex();

        // Если элемент был выбран, тогда отобразить выбор.
        if(idx != -1)
          jlab.setText("Current selection: " + cities[idx]);
                      // текущий выбор
        else  // В противном случае запросить выбор заново.
          jlab.setText("Choose a City");
      }
    });

    // Добавить список и метку в панель содержимого.
    jfrm.add(jscrlp);
    jfrm.add(jlab);

    // Отобразить фрейм.
    jfrm.setVisible(true);
  }
  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JListDemo();
        }
      }
    );
  }
}
```

Вывод программы `JListDemo` представлен на рис. 33.9.

Рис. 33.9. Вывод программы `JListDemo`

JComboBox

С помощью класса `JComboBox` в Swing можно создавать *комбинированный список* (сочетание текстового поля и раскрывающегося списка). Комбинированный список обычно отображает один элемент, но также способен показывать раскрывающийся список, позволяющий выбрать другую запись. Кроме того, можно создать комбинированный список, который позволяет вводить выбор в текстовом поле.

В прошлом элементы в `JComboBox` были представлены как ссылки `Object`. Тем не менее, начиная с версии JDK 7, класс `JComboBox` стал обобщенным и получил такое объявление:

```
class JComboBox<E>
```

Здесь `E` представляет тип элементов в комбинированном списке.

В примере используется следующий конструктор класса `JComboBox`:

```
JComboBox(E[] items)
```

В `items` указывается массив, который инициализирует комбинированный список. Доступны и другие конструкторы.

Класс `JComboBox` применяет модель `ComboBoxModel`. Изменяемые комбинированные списки (элементы которых могут быть модифицированы) используют модель `MutableComboBoxModel`.

В дополнение к передаче массива элементов, которые будут отображаться в раскрывающемся списке, элементы можно динамически добавлять в список вариантов с помощью метода `addItem()`:

```
void addItem(E obj)
```

В `obj` передается объект, который нужно добавить в комбинированный список. Данный метод должен применяться только с изменяемыми комбинированными списками.

Компонент `JComboBox` генерирует событие действия, когда пользователь выбирает элемент из списка. Кроме того, `JComboBox` генерирует событие элемента при изменении состояния выбора, что происходит, когда элемент выбирается или выбор его отменяется. Таким образом, изменение выбора означает, что произойдут два события элементов: одно для элемента с отме-

ненным выбором и одно для выбранного элемента. Часто достаточно просто прослушивать события действий, но для использования доступны оба вида событий.

Один из способов получения элемента, выбранного в списке, предусматривает вызов метода `getSelectedItem()` на комбинированном списке:

```
Object getSelectedItem()
```

Возвращаемое значение понадобится привести к типу объектов, хранящихся в списке.

В следующем примере демонстрируется применение комбинированного списка, который содержит элементы `"Hourglass"` (Песочные часы), `"Analog"` (Стрелочные часы), `"Digital"` (Цифровые часы) и `"Stopwatch"` (Секундомер). Когда часы выбраны, метка на основе значка обновляется с целью их отображения. Как видите, для использования этого мощного компонента требуется совсем мало кода.

```
// Демонстрация использования JComboBox.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JComboBoxDemo {

  String[] timepieces = { "Hourglass", "Analog", "Digital", "Stopwatch" };

  public JComboBoxDemo() {

    // Настроить JFrame.
    JFrame jfrm = new JFrame("JCombBoxDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(400, 250);

    // Создать комбинированный список и добавить его в панель содержимого.
    JComboBox<String> jcb = new JComboBox<String>(timepieces);
    jfrm.add(jcb);

    // Создать метку и добавить ее в панель содержимого.
    JLabel jlab = new JLabel(new ImageIcon("hourglass.png"));
    jfrm.add(jlab);

    // Обработать события выбора.
    jcb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        String s = (String) jcb.getSelectedItem();
        jlab.setIcon(new ImageIcon(s + ".png"));
      }
    });

    // Отобразить фрейм.
    jfrm.setVisible(true);
  }

  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(
```

```
      new Runnable() {
        public void run() {
          new JComboBoxDemo();
        }
      }
    );
  }
}
```

Вывод программы JComboBoxDemo показан на рис. 33.10.

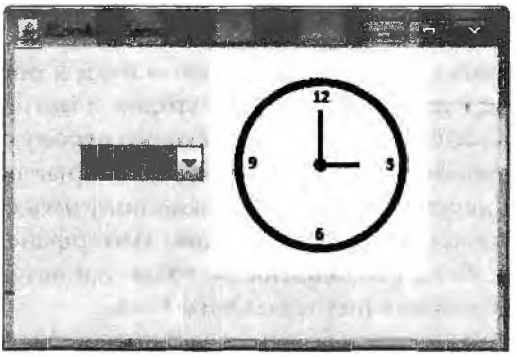


Рис. 33.10. Вывод программы JComboBoxDemo

Деревья

Дерево — это компонент, который обеспечивает иерархическое представление данных. В таком представлении пользователь имеет возможность разворачивать или сворачивать отдельные поддеревья. Деревья реализованы в Swing посредством класса JTree. Вот несколько форм его конструктора:

```
JTree(Object[] obj)
JTree(Vector<?> v)
JTree(TreeNode tn)
```

Первая форма строит дерево из элементов массива obj, вторая форма — из элементов вектора v, а третья форма — на основе корневого узла tn.

Хотя класс JTree расположен в пакете javax.swing, его поддерживающие классы и интерфейсы находятся в пакете javax.swing.tree. Причина в том, что количество классов и интерфейсов, необходимых для поддержки JTree, довольно велико.

Класс JTree полагается на две модели: TreeModel и TreeSelectionModel. Компонент JTree генерирует множество событий, но к деревьям имеют отношение три события: TreeExpansionEvent, TreeSelectionEvent и TreeModelEvent. Событие TreeExpansionEvent происходит, когда узел разворачивается или сворачивается. Событие TreeSelectionEvent генерируется, когда пользователь выбирает или отменяет выбор узла в дереве. Событие TreeModelEvent инициируется при изменении данных или структуры дерева. Прослушивателями упомянутых событий являются соответственно

TreeExpansionListener, TreeSelectionListener и TreeModelListener. Классы событий и интерфейсы прослушивателей для деревьев упакованы в javax.swing.event. В рассматриваемом далее примере обрабатывается событие TreeSelectionEvent. Для его прослушивания необходимо реализовать интерфейс TreeSelectionListener, в котором определен только один метод по имени valueChanged(), который принимает объект TreeSelectionEvent. Получить путь к выбранному объекту можно, вызвав метод getPath() на объекте события:

```
TreePath getPath()
```

Он возвращает объект TreePath, описывающий путь к измененному узлу. Класс TreePath инкапсулирует информацию о пути к определенному узлу в дереве и предоставляет несколько конструкторов и методов. В книге применяется только метод toString(), возвращающий строку с описанием пути.

В интерфейсе TreeNode объявлены методы, которые позволяют получить информацию об узле дерева. Например, можно получить ссылку на родительский узел или перечисление дочерних узлов. Интерфейс MutableTreeNode расширяет TreeNode. В нем объявлены методы для вставки и удаления дочерних узлов либо изменения родительского узла.

Класс DefaultMutableTreeNode реализует интерфейс MutableTreeNode и представляет узел в дереве. Ниже показан один из его конструкторов:

```
DefaultMutableTreeNode(Object obj)
```

В obj передается объект, который должен быть заключен в этот узел дерева. У нового узла дерева нет родителя или потомка.

Для создания иерархии узлов дерева предназначен метод add() класса DefaultMutableTreeNode со следующей сигнатурой:

```
void add(MutableTreeNode child)
```

В child указывается изменяемый узел дерева, который должен быть добавлен в качестве дочернего к текущему узлу.

Компонент JTree не предоставляет никаких собственных возможностей прокрутки. Взамен JTree обычно помещается внутрь JScrollPane. Таким образом, крупное дерево можно прокручивать в меньшем окне просмотра.

Использовать дерево можно с применением описанных далее шагов.

1. Создать экземпляр JTree.
2. Создать экземпляр JScrollPane и указать дерево в качестве объекта, подлежащего прокрутке.
3. Добавить панель прокрутки в панель содержимого.

В приведенном ниже примере демонстрируется создание дерева и обработка выбора. Сначала создается экземпляр DefaultMutableTreeNode с меткой "Options" (Параметры). Он будет верхним узлом иерархии дерева. Затем создаются дополнительные узлы дерева, которые с использованием метода add() присоединяются к дереву. Конструктору JTree в качестве аргумента

передается ссылка на верхний узел дерева. Далее дерево предоставляется в аргументе конструктору JScrollPane и созданная панель прокрутки добавляется в панель содержимого. Наконец, создается метка, которая тоже добавляется в панель содержимого. В ней будет отображаться выбор, произведенный в дереве. Чтобы получать события выбора в дереве, для дерева регистрируется прослушиватель TreeSelectionListener. Внутри метода valueChanged() получается и отображается путь к текущему выбору.

```
// Демонстрация использования JTree.

import java.awt.*;
import javax.swing.event.*;
import javax.swing.*;
import javax.swing.tree.*;

public class JTreeDemo {
  public JTreeDemo() {
    // Настроить JFrame. Использовать стандартную компоновку BorderLayout.
    JFrame jfrm = new JFrame("JTreeDemo");
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(200, 250);

    // Создать верхний узел дерева.
    DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");

    // Создать поддерево "A".
    DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
    top.add(a);
    DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
    a.add(a1);
    DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
    a.add(a2);

    // Создать поддерево "B".
    DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
    top.add(b);
    DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
    b.add(b1);
    DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
    b.add(b2);
    DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
    b.add(b3);

    // Создать дерево.
    JTree tree = new JTree(top);

    // Добавить дерево в панель прокрутки.
    JScrollPane jsp = new JScrollPane(tree);

    // Добавить панель прокрутки в панель содержимого.
    jfrm.add(jsp);

    // Добавить метку в панель содержимого.
    JLabel jlab = new JLabel();
    jfrm.add(jlab, BorderLayout.SOUTH);

    // Обработать события выбора в дереве.
    tree.addTreeSelectionListener(new TreeSelectionListener() {
```

```
      public void valueChanged(TreeSelectionEvent tse) {
        jlab.setText("Selection is " + tse.getPath());
      }
    });
    // Отобразить фрейм.
    jfrm.setVisible(true);
  }
  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JTreeDemo();
        }
      }
    );
  }
}
```

Вывод программы `JTreeDemo` представлен на рис. 33.11.

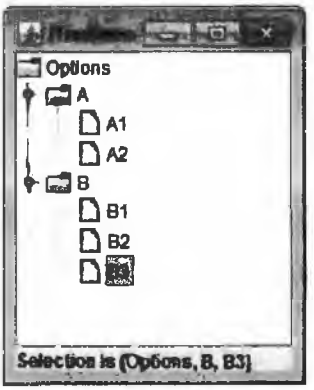


Рис. 33.11. Вывод программы `JTreeDemo`

Строка, представленная в текстовом поле, описывает путь от верхнего узла дерева до выбранного узла.

JTable

Компонент `JTable` отображает строки и колонки данных. Перетаскивая границы колонок с помощью мыши, можно изменять их размеры. Допускается также перетаскивать целую колонку на новое место. В зависимости от конфигурации можно выбирать строку, колонку или ячейку в таблице и модифицировать данные в ячейке. `JTable` — сложный компонент, предлагающий намного больше возможностей, чем можно здесь обсудить. (Вероятно, это самый сложный компонент Swing.) Однако стандартная конфигурация `JTable` все же обеспечивает солидную функциональность, которой легко пользоваться, особенно если просто нужно отображать данные в табличном формате.

Предлагаемый здесь краткий обзор дает общее представление о таком мощном компоненте.

Как и JTree, класс JTable имеет множество связанных с ним классов и интерфейсов. Они находятся в пакете javax.swing.table.

Концептуально компонент JTable довольно прост. Он состоит из одного или нескольких колонок информации. В верхней части каждой колонки имеется заголовок. Вдобавок к описанию данных в колонке заголовок также предлагает механизм, с помощью которого пользователь может изменять размер колонки либо ее расположение в таблице. Компонент JTable не предоставляет никаких собственных возможностей прокрутки, поэтому он обычно заключается в JScrollPane.

В классе JTable определено несколько конструкторов и вот тот, который применяется здесь:

```
JTable(Object[][] data, Object[] colHeads)
```

В data указывается двумерный массив представляемой информации, а в colHeads — одномерный массив с заголовками колонок.

Класс JTable опирается на три модели. Первая — табличная модель, которая определяется интерфейсом TableModel и отвечает за то, что связано с отображением данных в двумерном формате. Вторая — модель колонки таблицы, которая представлена интерфейсом TableColumnModel. С учетом того, что компонент JTable определяется в терминах колонок, именно TableColumnModel устанавливает характеристики колонки. Первые две модели упакованы в javax.swing.table. Третья модель определяет, каким образом выбираются элементы, и она указана посредством интерфейса ListSelectionModel, который был описан при обсуждении JList.

Компонент JTable способен генерировать несколько событий. Двумя наиболее важными для работы с таблицей являются ListSelectionEvent и TableModelEvent. Событие ListSelectionEvent генерируется, когда пользователь выбирает что-то в таблице. По умолчанию компонент JTable позволяет выбирать одну или несколько полных строк. Такое поведение можно изменить, чтобы разрешить выбор одной или нескольких колонок или одной или нескольких ячеек. Событие TableModelEvent инициируется, когда данные таблицы каким-либо образом изменяются. Обработка этих событий требует немного больше работы, чем обработка событий, генерируемых ранее описанными компонентами, и выходит за рамки настоящей книги. Тем не менее, если вы просто хотите использовать JTable для отображения данных (как в следующем примере), то обрабатывать какие-либо события не придется.

Ниже перечислены шаги, необходимые для настройки простого компонента JTable, который можно применять для отображения данных.

1. Создать экземпляр JTable.
2. Создать экземпляр JScrollPane, указав таблицу в качестве прокручиваемого объекта.
3. Добавить панель прокрутки в панель содержимого.

В следующем примере показано, как создать и использовать простую таблицу. Для заголовков колонок создается одномерный массив строк colHeads, а для ячеек — двумерный массив строк data. Легко заметить, что каждый элемент массива представляет собой массив из трех строк. Массивы data и colHeads передаются конструктору JTable. Таблица добавляется в панель прокрутки, которая добавляется в панель содержимого. В таблице отображаются данные из массива data. Стандартная конфигурация таблицы также позволяет редактировать содержимое ячеек. Изменения влияют на лежащий в основе массив, которым в данном случае является data.

```
// Демонстрация использования JTable.

import java.awt.*;
import javax.swing.*;

public class JTableDemo {

  // Инициализировать заголовки колонок.
  String[] colHeads = { "Name", "Extension", "ID#" };

  // Инициализировать данные.
  Object[][] data = {
    { "Gail", "4567", "865" },
    { "Ken", "7566", "555" },
    { "Viviane", "5634", "587" },
    { "Melanie", "7345", "922" },
    { "Anne", "1237", "333" },
    { "John", "5656", "314" },
    { "Matt", "5672", "217" },
    { "Claire", "6741", "444" },
    { "Erwin", "9023", "519" },
    { "Ellen", "1134", "532" },
    { "Jennifer", "5689", "112" },
    { "Ed", "9030", "133" },
    { "Helen", "6751", "145" }
  };

  public JTableDemo() {

    // Настроить JFrame. Использовать стандартную компоновку BorderLayout.
    JFrame jfrm = new JFrame("JTableDemo");
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(300, 300);

    // Создать таблицу.
    JTable table = new JTable(data, colHeads);

    // Добавить таблицу в панель прокрутки.
    JScrollPane jsp = new JScrollPane(table);

    // Добавить панель прокрутки в панель содержимого.
    jfrm.add(jsp);

    // Отобразить фрейм.
    jfrm.setVisible(true);
  }

  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
```

```
    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JTableDemo();
        }
      }
    );
  }
}
```

Вывод программы JTableDemo показан на рис. 33.12.

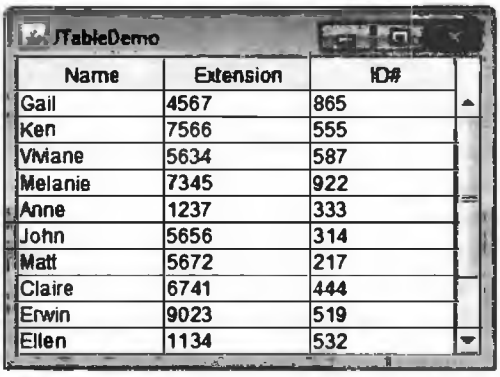


Рис. 33.12. Вывод программы JTableDemo

ГЛАВА

34 Введение в меню Swing

В настоящей главе представлен еще один фундаментальный аспект графических пользовательских интерфейсов Swing: меню. Меню являются неотъемлемой частью многих приложений, поскольку они представляют для пользователя функциональность программы. Из-за их важности в Swing обеспечивается обширная поддержка меню. Именно в этой области становится очевидной вся мощь Swing.

Система меню Swing поддерживает перечисленные ниже ключевые элементы.

- Панель меню, представляющая собой главное мсеню для приложения.
- Стандартное меню, которое может содержать либо выбираемые пункты, либо другие меню (подменю).
- Всплывающее меню, которое обычно активизируется щелчком правой кнопкой мыши.
- Панель инструментов, которая обеспечивает быстрый доступ к функциональным возможностям приложения, часто параллельно пунктам меню.
- Действие, которое дает возможность одному объекту управлять двумя и более компонентами. Действия обычно используются с меню и панелями инструментов.

Меню Swing также поддерживают клавиатурные сочетания, которые позволяют выбирать пункты меню без активизации самого меню, и мнемонические символы, позволяющие выбирать пункты меню с помощью клавиатуры после отображения меню.

Основы меню

Система меню Swing поддерживается группой связанных классов. В табл. 34.1 кратко описаны классы, применяемые в этой главе, которые образуют ядро системы меню. Хотя поначалу меню Swing могут показаться немного запутанными, они довольно просты в использовании. При желании

Swing допускает высокую степень настройки; однако обычно классы меню будут применяться в том виде, как есть, поскольку они поддерживают все самые востребованные варианты. Например, в меню можно легко добавлять изображения и клавиатурные сочетания.

Таблица 34.1. Основные классы меню Swing

Класс	Описание
`JMenuBar`	Объект, который содержит меню верхнего уровня для приложения
`JMenu`	Стандартное меню, состоящее из одного или большего количества объектов `JMenuItem`
`JMenuItem`	Объект, которым заполняется меню
`JCheckBoxMenuItem`	Пункт меню с флажком
`JRadioButtonMenuItem`	Пункт меню с переключателем
`JSeparator`	Визуальный разделитель между пунктами меню
`JPopupMenu`	Меню, которое обычно активизируется щелчком правой кнопкой мыши

Ниже приведен краткий обзор того, как классы сочетаются друг с другом. Чтобы создать меню верхнего уровня для приложения, сначала создается объект класса `JMenuBar`, который является своего рода контейнером для меню. К экземпляру `JMenuBar` будут добавляться экземпляры `JMenu`. Каждый объект `JMenu` определяет меню, т.е. содержит один или несколько выбираемых пунктов. Пункты, отображаемые `JMenu`, представляют собой объекты `JMenuItem`. Таким образом, объект `JMenuItem` определяет вариант, который может выбрать пользователь.

В качестве альтернативы или дополнения панели меню можно также создавать автономные всплывающие меню, для чего понадобится создать объект типа `JPopupMenu` и затем добавить к нему объекты `JMenuItem`. Всплывающее меню обычно активизируется щелчком правой кнопкой мыши, когда ее указатель находится на компоненте, для которого было определено всплывающее меню.

Помимо "стандартных" пунктов в меню можно также включать пункты с флажками и переключателями. Для создания пункта меню с флажком используется класс `JCheckBoxMenuItem`, а пункта меню с переключателем — класс `JRadioButtonMenuItem`. Оба класса расширяют `JMenuItem`. Их можно применять в стандартных и всплывающих меню.

Класс `JToolBar` создает автономный компонент, связанный с меню. Он часто используется для обеспечения быстрого доступа к функциональности, содержащейся в меню приложения. Например, панель инструментов может обеспечивать быстрый доступ к командам форматирования, которые поддерживаются текстовым процессором.

JSeparator — удобный класс для создания разделительной линии в меню.

Касательно меню Swing важно понимать, что каждый пункт меню расширяет AbstractButton. Вспомните, что AbstractButton также выступает в качестве суперкласса для всех кнопочных компонентов Swing вроде JButton. Таким образом, все пункты меню по существу являются кнопками. Очевидно, что в случае применения в меню они не выглядят похожими на кнопки, но во многих отношениях они действуют как кнопки. Например, выбор пункта меню генерирует событие действия подобно нажатию кнопки.

Еще один важный момент заключается в том, что JMenuItem — суперкласс JMenu. Это позволяет создавать подменю, которые представляют собой меню внутри меню. Чтобы создать подменю, сначала создается и заполняется объект JMenu, после чего он добавляется к другому объекту JMenu. Такой процесс демонстрируется в следующем разделе.

Как мимоходом упоминалось ранее, при выборе пункта меню генерируется событие действия. Строка с командой действия, связанная с данным событием, по умолчанию будет именем выбора в меню. Таким образом, для определения выбранного пункта можно проанализировать команду действия. Конечно, для обработки событий действий каждого пункта меню можно также использовать отдельные анонимные внутренние классы или лямбда-выражения. В таком случае выбор в меню уже известен и нет необходимости проверять строку с командой действия, чтобы выяснить, какой пункт был выбран.

Меню также способны генерировать другие типы событий. Например, каждый раз, когда меню активизируется, выбирается или его выбор отменяется, генерируется событие MenuEvent, которое можно прослушивать с помощью MenuListener. Другие события, связанные с меню, включают MenuKeyEvent, MenuDragMouseEvent и PopupMenuEvent. Тем не менее, во многих случаях нужно отслеживать только события действий, которые и будут применяться в этой главе.

Обзор JMenuBar, JMenu и JMenuItem

Прежде чем приступать к созданию меню, необходимо обсудить три ключевых классах меню: JMenuBar, JMenu и JMenuItem. Они формируют минимальный набор классов, необходимых для построения главного меню приложения. Классы JMenu и JMenuItem также используются всплывающими меню. Таким образом, указанные классы составляют основу системы меню.

JMenuBar

Как уже упоминалось, класс JMenuBar по существу является контейнером для меню. Подобно всем компонентам он унаследован от класса JComponent (который унаследован от Container и Component). Класс JMenuBar имеет только один стандартный конструктор. Поэтому изначально панель меню будет пустой, и перед применением ее нужно будет заполнить. Каждое приложение имеет одну и только одну панель меню.

В классе JMenuBar определено несколько методов, но чаще всего будет использоваться только один: add(). Метод add() добавляет экземпляр JMenu в панель меню:

```
JMenu add(JMenu menu)
```

В menu указывается экземпляр JMenu, подлежащий добавлению в панель меню. Метод возвращает ссылку на меню. Меню располагаются в панели слева направо в том порядке, в каком они добавлялись. Для добавления меню в определенное место предназначена следующая версия add(), унаследованная из Container:

```
Component add(Component menu, int idx)
```

Меню menu добавляется по индексу idx. Индексация начинается с 0, где 0 соответствует крайнему слева меню.

В некоторых случаях может потребоваться удалить меню, которое больше не нужно. Для этого необходимо вызвать метод remove(), унаследованный из Container. Он имеет две формы:

```
void remove(Component menu)
void remove(int idx)
```

В menu передается ссылка, а в idx — индекс удаляемого меню. Индексация начинается с 0.

Временами полезен метод getMenuCount():

```
int getMenuCount()
```

Он возвращает количество пунктов, содержащихся в панели меню.

В JMenuBar определен ряд других методов, которые могут оказаться полезными в специализированных приложениях. Например, с помощью вызова getSubElements() можно получить массив ссылок на меню в панели, а посредством вызова isSelected() — выяснить, выбрано ли меню.

После создания и заполнения панели меню вызовом setJMenuBar() на экземпляре JFrame она добавляется в JFrame. (Панели меню *не* добавляются в панель содержимого.) Метод setJMenuBar() показан ниже:

```
void setJMenuBar(JMenuBar mb)
```

В mb указывается ссылка на панель меню. Панель меню будет отображаться в позиции, определяемой внешним видом — обычно в верхней части окна.

JMenu

Класс JMenu инкапсулирует меню, которое заполняется экземплярами JMenuItem. Как уже упоминалось, он является производным от JMenuItem, т.е. один экземпляр JMenu может быть выбором в другом экземпляре JMenu, позволяя одному меню быть подменю другого. В классе JMenu определено несколько конструкторов. Скажем, вот конструктор, который применяется в примерах далее в главе:

```
JMenu(String name)
```

Он создает меню с заголовком, указанным в name. Разумеется, назначать меню имя вовсе не обязательно. Для создания неименованного меню можно использовать стандартный конструктор:

```
JMenu()
```

Поддерживаются и другие конструкторы. В каждом случае меню остается пустым до тех пор, пока в него не будут добавлены пункты меню.

В классе JMenu определено множество методов. Ниже кратко описаны самые распространенные из них. Для добавления пункта в меню предназначен метод add(), который имеет несколько форм, в том числе следующие две:

```
JMenuItem add(JMenuItem item)
Component add(Component item, int idx)
```

В item передается добавляемый пункт меню. Первая форма добавляет пункта в конец меню. Вторая форма добавляет пункт по индексу, указанному в idx. Как и ожидалось, индексация начинается с 0. Обе формы возвращают ссылку на добавленный пункт. Интересно отметить, что для добавления пунктов в меню можно также применять метод insert().

Вызвав метод addSeparator(), в меню можно добавить разделитель (объект типа JSeparator):

```
void addSeparator()
```

Разделитель добавляется в конец меню. Для вставки разделителя в меню понадобится вызвать метод insertSeparator():

```
void insertSeparator(int idx)
```

В idx указывается индекс, начинающийся с 0, по которому будет добавлен разделитель.

Удалить пункт из меню можно посредством метода remove(). Вот две его формы:

```
void remove(JMenuItem menu)
void remove(int idx)
```

В menu передается ссылка, а в idx — индекс удаляемого пункта.

Получить количество пунктов в меню можно с помощью метода getMenuComponentCount():

```
int getMenuComponentCount()
```

Вызвав метод getMenuComponents(), можно получить массив пунктов меню:

```
Component[] getMenuComponents()
```

Метод возвращает массив, содержащий компоненты меню.

JMenuItem

Класс JMenuItem инкапсулирует пункт в меню, который может быть выбором, связанным с каким-то программным действием вроде Save (Сохранить) или Close (Закрыть), либо вызывать отображение подменю. Как уже упоми-

налось, класс JMenuItem является производным от AbstractButton, и каждый пункт меню можно рассматривать как особый вид кнопки. Поэтому при выборе пункта меню генерируется событие действия, что похоже на то, как компонент JButton инициирует событие действия при нажатии. В классе JMenuItem определено множество конструкторов. Ниже приведены те, которые используются в главе:

```
JMenuItem(String name)
JMenuItem(Icon image)
JMenuItem(String name, Icon image)
JMenuItem(String name, int mnem)
JMenuItem(Action action)
```

Первый конструктор создает пункт меню с именем, заданным в name. Второй создает пункт меню, отображающий изображение, указанное в image. Третий создает пункт меню с именем name и изображением image. Четвертый создает пункт меню с именем name и мнемоническим символом, заданным в mnem. Мнемонический символ позволяет выбирать пункт из меню, нажимая указанную клавишу. Последний конструктор создает пункт меню с применением информации, заданной в action. Вдобавок поддерживается стандартный конструктор.

Поскольку пункты меню унаследованы от AbstractButton, имеется доступ к функциональности, предоставляемой классом AbstractButton. Одним из таких методов, часто используемых вместе с меню, является setEnabled(), который можно применять для включения или отключения пункта меню:

```
void setEnabled(boolean enable)
```

Если enable равно true, тогда пункт меню включается, а если false, то пункт отключается и не может быть выбран.

Создание главного меню

Традиционно наиболее часто используется *главное меню*. Оно определено панелью меню и открывает доступ ко всей (или почти всей) функциональности приложения. К счастью, Swing упрощает создание и управление главным меню. В этом разделе показано, как создать основное главное меню, а в последующих разделах — каким образом добавить к нему варианты выбора.

Главное меню строится за несколько шагов. Сначала необходимо создать объект JMenuBar, который будет содержать меню, после чего создать каждое меню, которое будет находиться в панели меню. Как правило, для построения меню понадобится создать объект JMenu и добавить к нему экземпляры JMenuItem. Готовое меню нужно добавить в панель меню, которую затем добавить во фрейм вызовом setJMenuBar(). Наконец, для каждого пункта меню должен быть добавлен прослушиватель событий действий, который обрабатывает событие действия, инициируемое при выборе пункта меню.

Лучше понять процесс создания и управления меню поможет пример. В следующей программе создается простая панель с тремя меню. Первое

представляет собой стандартное меню File (Файл), которое содержит пункты Open (Открыть), Close (Закрыть), Save (Сохранить) и Exit (Выход). Второе меню, Options (Параметры), содержит два подменю — Colors (Цвета) и Priority (Приоритет). Третье меню, Help (Справка), состоит из одного пункта — About (О программе). Когда пункт меню выбран, имя выбора отображается внутри метки в панели содержимого. Пример вывода показан на рис. 34.1.

```
// Демонстрация простого главного меню.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MenuDemo implements ActionListener {
  JLabel jlab;

  MenuDemo() {
    // Создать контейнер JFrame.
    JFrame jfrm = new JFrame("Menu Demo");

    // Установить диспетчер компоновки FlowLayout.
    jfrm.setLayout(new FlowLayout());

    // Назначить фрейму начальные размеры.
    jfrm.setSize(220, 200);

    // Прекращать работу при закрытии пользователем приложения.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создать метку, в которой будет отображаться выбор меню.
    jlab = new JLabel();

    // Создать панель меню.
    JMenuBar jmb = new JMenuBar();

    // Создать меню File.
    JMenu jmFile = new JMenu("File");
    JMenuItem jmiOpen = new JMenuItem("Open");
    JMenuItem jmiClose = new JMenuItem("Close");
    JMenuItem jmiSave = new JMenuItem("Save");
    JMenuItem jmiExit = new JMenuItem("Exit");
    jmFile.add(jmiOpen);
    jmFile.add(jmiClose);
    jmFile.add(jmiSave);
    jmFile.addSeparator();
    jmFile.add(jmiExit);
    jmb.add(jmFile);

    // Создать меню Options.
    JMenu jmOptions = new JMenu("Options");

    // Создать подменю Colors.
    JMenu jmColors = new JMenu("Colors");
    JMenuItem jmiRed = new JMenuItem("Red");
    JMenuItem jmiGreen = new JMenuItem("Green");
    JMenuItem jmiBlue = new JMenuItem("Blue");
    jmColors.add(jmiRed);
    jmColors.add(jmiGreen);
    jmColors.add(jmiBlue);
    jmOptions.add(jmColors);
```

```
    // Создать подменю Priority.
    JMenu jmPriority = new JMenu("Priority");
    JMenuItem jmiHigh = new JMenuItem("High");
    JMenuItem jmiLow = new JMenuItem("Low");
    jmPriority.add(jmiHigh);
    jmPriority.add(jmiLow);
    jmOptions.add(jmPriority);
    // Создать пункт меню Reset.
    JMenuItem jmiReset = new JMenuItem("Reset");
    jmOptions.addSeparator();
    jmOptions.add(jmiReset);
    // Добавить меню Options в панель меню.
    jmb.add(jmOptions);
    // Создать меню Help.
    JMenu jmHelp = new JMenu("Help");
    JMenuItem jmiAbout = new JMenuItem("About");
    jmHelp.add(jmiAbout);
    jmb.add(jmHelp);
    // Добавить прослушиватели событий действий для пунктов меню.
    jmiOpen.addActionListener(this);
    jmiClose.addActionListener(this);
    jmiSave.addActionListener(this);
    jmiExit.addActionListener(this);
    jmiRed.addActionListener(this);
    jmiGreen.addActionListener(this);
    jmiBlue.addActionListener(this);
    jmiHigh.addActionListener(this);
    jmiLow.addActionListener(this);
    jmiReset.addActionListener(this);
    jmiAbout.addActionListener(this);
    // Добавить метку в панель содержимого.
    jfrm.add(jlab);
    // Добавить панель меню во фрейм.
    jfrm.setJMenuBar(jmb);
    // Отобразить фрейм.
    jfrm.setVisible(true);
  }
  // Обработать события действий для пунктов меню.
  public void actionPerformed(ActionEvent ae) {
    // Получить команду действия из выбора меню.
    String comStr = ae.getActionCommand();
    // Если пользователь выбрал Exit, тогда завершить программу.
    if(comStr.equals("Exit")) System.exit(0);
    // Иначе отобразить выбор.
    jlab.setText(comStr + " Selected");
  }
  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new MenuDemo();
      }
    });
  }
}
```

Рис. 34.1. Пример вывода программы `MenuDemo`

Рассмотрим, каким образом создаются меню в этой программе, и первым обсудим конструктор `MenuDemo`. Он начинается с создания экземпляра `JFrame` и настройки его диспетчера компоновки, размеров и стандартной операции закрытия. (Такие операции были описаны в главе 32.) Затем создается экземпляр `JLabel`, который будет применяться для отображения выбора меню. Далее создается панель меню, и ссылка на нее присваивается `jmb` посредством следующего оператора:

```
// Создать панель меню.
JMenuBar jmb = new JMenuBar();
```

С помощью приведенной ниже кодовой последовательности создается меню File, ссылка на которое присваивается переменной `jmFile`, и его пункты меню:

```
// Создать меню File.
JMenu jmFile = new JMenu("File");
JMenuItem jmiOpen = new JMenuItem("Open");
JMenuItem jmiClose = new JMenuItem("Close");
JMenuItem jmiSave = new JMenuItem("Save");
JMenuItem jmiExit = new JMenuItem("Exit");
```

Названия Open, Close, Save и Exit будут отображаться в качестве вариантов выбора в меню. Затем пункты меню добавляются в меню File в такой последовательности:

```
jmFile.add(jmiOpen);
jmFile.add(jmiClose);
jmFile.add(jmiSave);
jmFile.addSeparator();
jmFile.add(jmiExit);
```

В заключение меню File добавляется в панель меню:

```
jmb.add(jmFile);
```

После завершения предыдущей кодовой последовательности панель меню будет иметь одно меню: File. Меню File будет содержать четыре пункта в следующем порядке: Open, Close, Save и Exit. Однако обратите внимание, что перед пунктом Exit был добавлен разделитель, который визуально отделяет Exit от трех предшествующих пунктов.

Меню Options построено с использованием того же базового процесса, что и меню File. Тем не менее, меню Options состоит из двух подменю, Colors и Priority, и пункта Reset (Сброс). Подменю создаются по отдельности и добавляются в меню Options. Пункт Reset добавляется последним. Затем меню Options добавляется в панель меню. Процесс построения меню Help аналогичен.

Обратите внимание, что класс MenuDemo реализует интерфейс ActionListener, а события действий, сгенерированные при выборе меню, обрабатываются методом actionPerformed(), который определен в MenuDemo. Поэтому программа добавляет this в качестве прослушивателя событий действий для пунктов меню. Кроме того, к пунктам Colors и Priority прослушиватели не добавляются, потому что на самом деле они не являются выборами, а просто активизируют подменю. Наконец, панель меню добавляется во фрейм посредством следующей строки:

```
jfrm.setJMenuBar(jmb);
```

Как уже упоминалось, панели меню добавляются не в панель содержимого, а непосредственно к компоненту JFrame.

Метод actionPerformed() обрабатывает события действий, генерируемые меню. Он получает строку с командой действия, связанную с выбором, вызывая метод getActionCommand() на объекте события. Ссылка на эту строку сохраняется в comStr. Затем команда действия проверяется на равенство "Exit":

```
if(comStr.equals("Exit")) System.exit(0);
```

Если командой действия является "Exit", тогда программа завершается за счет вызова метода System.exit(), который приводит к немедленному прекращению работы программы и передает свой аргумент в виде кода состояния вызывающему процессу — обычно операционной системе. По соглашению нулевой код состояния означает нормальное окончание, а все остальные указывают на аварийное завершение программы. Для всех остальных пунктов меню отображается выбор.

На данном этапе можете немного поэкспериментировать с программой MenuDemo. Попробуйте добавить еще одно меню или дополнительные пункты в существующее меню. Важно, чтобы вы поняли базовые концепции меню, прежде чем двигаться дальше, поскольку эта программа будет развиваться на протяжении всей главы.

Добавление мнемонических символов и клавиатурных сочетаний к пунктам меню

Меню, созданное в предыдущем примере, полностью функционально, но его можно улучшить. В реальных приложениях меню обычно включает поддержку клавиш быстрого доступа, т.к. они дают опытному пользователю возможность оперативно выбирать пункты меню. Клавиши быстрого доступа бывают двух видов: мнемонические символы и клавиатурные сочетания.

Применительно к меню *мнемонический символ* определяет клавишу, которая позволяет выбрать пункт из активного меню, нажав клавишу, т.е. использовать клавиатуру для выбора пункта из меню, которое уже отображается. *Клавиатурное сочетание* — это клавиша, которая позволяет выбирать пункт меню без необходимости предварительной активизации меню.

Мнемонический символ может быть указан как для объектов JMenuItem, так и для объектов JMenu. Установить мнемонический символ для JMenuItem можно двумя способами. Во-первых, его можно указать, когда объект создается с помощью следующего конструктора:

```
JMenuItem(String name, int mnem)
```

В name передается имя, а в mnem — мнемонический символ. Во-вторых, мнемонический символ можно установить с помощью метода setMnemonic(). Чтобы указать мнемонический символ для JMenu, понадобится вызвать setMnemonic(). Метод setMnemonic() унаследован обоими классами от AbstractButton и показан далее:

```
void setMnemonic(int mnem)
```

В mnem задается мнемонический символ. Он должен быть одной из констант, определенных в java.awt.event.KeyEvent, скажем, KeyEvent.VK_F или KeyEvent.VK_Z. (Существует еще одна версия setMnemonic(), которая принимает аргумент char, но она считается устаревшей.) Мнемонические символы не чувствительны к регистру, поэтому в случае VK_A допускается вводить либо a, либо A.

По умолчанию первая совпадающая буква в пункте меню будет подчеркнута. При желании подчеркнуть букву, отличающуюся от первого совпадения, необходимо передать индекс буквы в качестве аргумента метода setDisplayedMnemonicIndex(), который унаследован классами JMenu и JMenuItem от AbstractButton:

```
void setDisplayedMnemonicIndex(int idx)
```

В idx указывается индекс подчеркиваемой буквы.

С объектом JMenuItem может быть связано клавиатурное сочетание, которое задается вызовом метода setAccelerator():

```
void setAccelerator(KeyStroke ks)
```

В ks указывается клавиатурное сочетание, нажатие которого позволяет выбрать пункт меню. Класс KeyStroke содержит несколько фабричных методов, предназначенных для создания различных видов клавиатурных сочетаний. Вот три примера:

```
static KeyStroke getKeyStroke(char ch)
static KeyStroke getKeyStroke(Character ch, int modifier)
static KeyStroke getKeyStroke(int ch, int modifier)
```

В ch передается клавиатурное сочетание. В первой форме ch указывается как значение char, во второй — как объект типа Character, а в третьей — как значение описанного ранее типа KeyEvent.

Значение modifier — это одно или несколько перечисленных далее констант, определенных в классе java.awt.event.InputEvent:

```
InputEvent.ALT_DOWN_MASK          InputEvent.ALT_GRAPH_DOWN_MASK
InputEvent.CTRL_DOWN_MASK         InputEvent.META_DOWN_MASK
InputEvent.SHIFT_DOWN_MASK
```

Таким образом, в случае передачи VK_A в ch и InputEvent.CTRL_DOWN_MASK в modifier клавиатурным сочетанием будет <Ctrl+A>.

Следующая кодовая последовательность добавляет мнемонические символы и клавиатурные сочетания в меню File, созданное программой MenuDemo в предыдущем разделе. После такого изменения меню File можно выбирать, нажимая <Alt+F>, а для выбора пунктов открывшегося меню можно применять мнемонические символы O, C, S и E. Кроме того, пункты меню File можно выбирать напрямую, нажимая <Ctrl+O>, <Ctrl+C>, <Ctrl+S> и <Ctrl+E>. На рис. 34.2 показано, как обновленное меню File выглядит после активизации.

```
// Создать меню File с мнемоническими символами и клавиатурными сочетаниями
JMenu jmFile = new JMenu("File");
jmFile.setMnemonic(KeyEvent.VK_F);

JMenuItem jmiOpen = new JMenuItem("Open", KeyEvent.VK_O);
jmiOpen.setAccelerator(
     KeyStroke.getKeyStroke(KeyEvent.VK_O, InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiClose = new JMenuItem("Close", KeyEvent.VK_C);
jmiClose.setAccelerator(
     KeyStroke.getKeyStroke(KeyEvent.VK_C, InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiSave = new JMenuItem("Save", KeyEvent.VK_S);
jmiSave.setAccelerator(
     KeyStroke.getKeyStroke(KeyEvent.VK_S, InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiExit = new JMenuItem("Exit", KeyEvent.VK_E);
jmiExit.setAccelerator(
     KeyStroke.getKeyStroke(KeyEvent.VK_E, InputEvent.CTRL_DOWN_MASK));
```

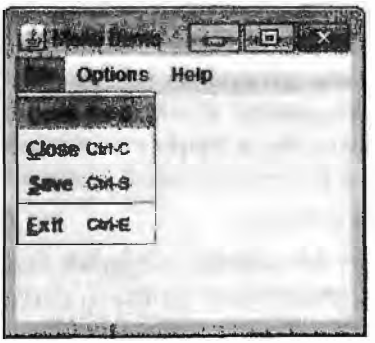


Рис. 34.2. Меню File после добавления мнемонических символов и клавиатурных сочетаний

Добавление изображений и всплывающих подсказок к пунктам меню

К пунктам меню можно добавлять изображения либо использовать изображения вместо текста. Самый простой способ добавить изображение — указать его при создании пункта меню с помощью одного из следующих форм конструкторов:

```
JMenuItem(Icon image)
JMenuItem(String name, Icon image)
```

Первая форма создает пункт меню, который отображает изображение, заданное в image. Вторая форма создает пункт меню с именем, указанным в name, и изображением, указанным в image. Например, вот как создать пункт меню About, ассоциированный с изображением:

```
ImageIcon icon = new ImageIcon("AboutIcon.gif");
JMenuItem jmiAbout = new JMenuItem("About", icon);
```

Рис. 34.3. Пункт меню About с добавленным значком

После такого добавления значок, заданный в icon, будет отображаться рядом с текстом "About" при отображении меню Help (рис. 34.3). Добавить значок к пункту меню можно и после его создания, вызвав метод setIcon(), который унаследован из AbstractButton. Посредством метода setHorizontalTextPosition() можно установить горизонтальное выравнивание изображения относительно текста.

Вызвав метод setDisabledIcon(), можно указать значок недоступности, который отображается, когда пункт меню отключен. Обычно когда пункт меню отключен, стандартный значок отображается серым цветом. Если указан значок недоступности, то именно он будет отображаться для отключенного пункта меню.

Всплывающая подсказка представляет собой небольшое сообщение, описывающее пункт меню. Она отображается автоматически, когда указатель мыши остается над пунктом меню в течение короткого времени. Чтобы добавить всплывающую подсказку к пункту меню, необходимо вызвать метод setToolTipText() на пункте, указав текст, который требуется отобразить:

```
void setToolTipText(String msg)
```

В данном случае msg — это строка, которая будет отображаться при активизации всплывающей подсказки. Скажем, следующий оператор создает всплывающую подсказку для пункта меню About:

```
jmiAbout.setToolTipText("Info about the MenuDemo program.");
```

Интересно отметить, что метод setToolTipText() унаследован классом JMenuItem из JComponent. Таким образом, всплывающую подсказку можно добавлять к другим типам компонентов, например, к кнопке. Имеет смысл опробовать это самостоятельно.

Использование JRadioButtonMenuItem и JCheckBoxMenuItem

Хотя типы пунктов меню, применяемые в предшествующих примерах, как правило, являются наиболее распространенными, в Swing определены еще два: флажки и переключатели. Такие пункты могут оптимизировать графический интерфейс, позволяя меню предоставлять функциональность, которая в противном случае потребовала бы дополнительных автономных компонентов. Кроме того, иногда помещение флажков или переключателей в меню кажется наиболее естественным местом для определенного набора функциональных средств. Какова бы ни была причина, Swing упрощает использование флажков и переключателей в меню, и здесь рассматриваются оба приема.

Для добавления флажка в меню нужно создать экземпляр класса JCheckBoxMenuItem, в котором определено несколько конструкторов. Вот тот, который применяется в главе:

```
JCheckBoxMenuItem(String name)
```

В name задается имя пункта. В начальном состоянии флажок не отмечен. Указать начальное состояние дает возможность следующий конструктор:

```
JCheckBoxMenuItem(String name, boolean state)
```

Если state равно true, тогда флажок изначально отмечен. В противном случае он не отмечен. Класс JCheckBoxMenuItem также предоставляет конструкторы, позволяющие указывать значок, например:

```
JCheckBoxMenuItem(String name, Icon icon)
```

В name передается имя пункта, а в icon — изображение, ассоциированное с пунктом. Изначально пункт меню не отмечен. Поддерживаются и другие конструкторы.

Флажки в меню работают аналогично автономным флажкам. Скажем, они генерируют события действий и события элементов при изменении их состояния. Флажки особенно полезны в меню, когда есть параметры, которые можно выбирать, и желательно визуально отображать их состояние "выбран" или "не выбран".

Для добавления переключателя в меню понадобится создать объект типа JRadioButtonMenuItem. Класс JRadioButtonMenuItem унаследован от JMenuItem и предлагает богатый набор конструкторов. Ниже перечислены конструкторы, которые используются в главе:

```
JRadioButtonMenuItem(String name)
JRadioButtonMenuItem(String name, boolean state)
```

Первый конструктор создает пункт меню с невыбранным переключателем и именем, переданным в name. Второй конструктор позволяет указать начальное состояние переключателя. Если state равно true, тогда переключатель изначально выбран. В противном случае он не выбран. Другие конструкторы позволяют указать значок, например:

```
JRadioButtonMenuItem(String name, Icon icon, boolean state)
```

Данный конструктор создает пункт меню с переключателем, имя которого передается в name, а изображение — в icon. Если state равно true, тогда переключатель изначально выбран. В противном случае он не выбран. Поддерживаются и несколько других конструкторов.

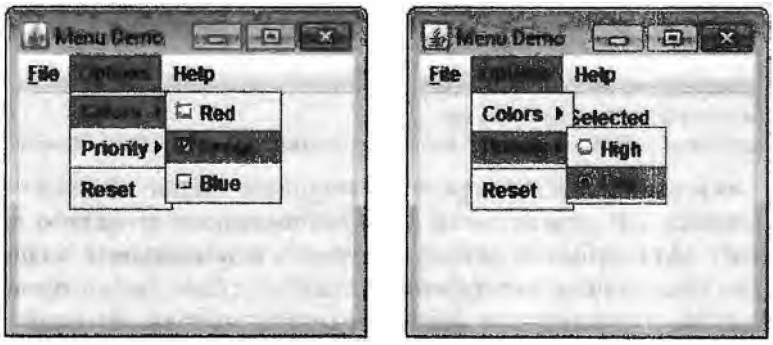
Экземпляр JRadioButtonMenuItem работает как отдельный переключатель, генерируя события элементов и действий. Подобно автономным переключателям для инициирования взаимоисключающего поведения при выборе переключатели в пунктах меню должны быть помещены в группу кнопок.

Поскольку классы JCheckBoxMenuItem и JRadioButtonMenuItem унаследованы от JMenuItem, каждый из них обладает всей функциональностью, обеспечиваемой JMenuItem. Помимо дополнительных возможностей флажков и переключателей они действуют и применяются аналогично остальным пунктам меню.

Чтобы опробовать пункты меню с флажками и переключателями, сначала необходимо удалить код, который создает меню Options в программе MenuDemo, после чего подставить следующую кодовую последовательность, в которой используются флажки для подменю Colors и переключатели для подменю Priority. После такой подстановки меню Options будет выглядеть так, как показано на рис. 34.4.

```
// Создать меню Options.
JMenu jmOptions = new JMenu("Options");

// Создать подменю Colors.
JMenu jmColors = new JMenu("Colors");

// Использовать флажки для цветов. Они позволяют
// пользователю выбирать более одного цвета.
JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");
JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");

jmColors.add(jmiRed);
jmColors.add(jmiGreen);
jmColors.add(jmiBlue);
jmOptions.add(jmColors);

// Создать подменю Priority.
JMenu jmPriority = new JMenu("Priority");

// Использовать переключатели для настройки приоритета. Они дают
// возможность показывать в меню применяемый приоритет и также
// гарантируют, что в любой момент времени может быть выбран один
// и только один приоритет. Обратите внимание, что изначально
// выбран переключатель High (Высокий).
```

```
JRadioButtonMenuItem jmiHigh =
  new JRadioButtonMenuItem("High", true);
JRadioButtonMenuItem jmiLow =
  new JRadioButtonMenuItem("Low");

jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

// Создать группу кнопок для пунктов меню с переключателями.
ButtonGroup bg = new ButtonGroup();
bg.add(jmiHigh); bg.add(jmiLow);

// Создать пункт меню Reset.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// В заключение добавить все меню Options в панель меню.
jmb.add(jmOptions);
```



а) б)

Рис. 34.4. Пункты меню с флажками (а) и переключателями (б)

Создание всплывающего меню

Популярной альтернативой или дополнением панели меню является всплывающее меню. Обычно всплывающее меню активизируется щелчком правой кнопкой мыши на компоненте. Всплывающие меню поддерживаются в Swing классом JPopupMenu, который имеет два конструктора. В главе применяется только стандартный конструктор:

```
JPopupMenu()
```

Он создает стандартное всплывающее меню. Другой конструктор позволяет указать заголовок для меню. В зависимости от внешнего вида этот заголовок будет отображаться или нет.

В целом всплывающие меню устроены так же, как и обычные меню. Первым делом нужно создать объект JPopupMenu, после чего добавить к нему пункты меню. Выбор пункта меню обрабатывается аналогично: путем прослушивания событий действий. Основное различие между всплывающим меню и обычным меню заключается в процессе активизации.

Активизация всплывающего меню требует выполнения трех шагов.

1. Зарегистрировать прослушиватель для событий мыши.
2. Отслеживать инициатор всплывающего меню внутри обработчика событий мыши.
3. При получении инициатора всплывающего меню отобразить всплывающее меню, вызвав метод `show()`.

Проанализируем описанные выше шаги.

Всплывающее меню обычно активизируется щелчком правой кнопкой мыши, когда указатель мыши находится над компонентом, для которого определено всплывающее меню. Таким образом, *инициатор всплывающего меню* обычно активизируется щелчком правой кнопкой мыши на компоненте с включенным всплывающим меню. Для прослушивания инициатора всплывающего меню необходимо реализовать интерфейс `MouseListener` и зарегистрировать прослушиватель, вызвав метод `addMouseListener()`. Как было описано в главе 25, в `MouseListener` определены следующие методы:

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me) void mouseReleased(MouseEvent me)
```

Два из них очень важны для всплывающего меню: `mousePressed()` и `mouseReleased()`. В зависимости от установленного внешнего вида любое из указанных двух событий может запускать всплывающее меню. По этой причине для реализации интерфейса `MouseListener` часто проще использовать класс `MouseAdapter` и переопределять методы `mousePressed()` и `mouseReleased()`.

В классе `MouseEvent` определено несколько методов, но при активизации всплывающего меню обычно требуются только четыре метода:

```
int getX()
int getY()
boolean isPopupTrigger()
Component getComponent()
```

Текущая позиция курсора мыши X,Y относительно источника события определяется вызовами `getX()` и `getY()`. Они применяются для указания верхнего левого угла всплывающего меню при его отображении. Метод `isPopupTrigger()` возвращает `true`, если событие мыши представляет инициатор всплывающего меню, или `false` в противном случае. Этот метод будет использоваться при выяснении, когда отображать всплывающее меню. Для получения ссылки на компонент, сгенерировавший событие мыши, понадобится вызвать метод `getComponent()`.

Чтобы действительно отобразить всплывающее меню, необходимо вызвать метод `show()`, определенный в `JPopupMenu`:

```
void show(Component invoker, int upperX, int upperY)
```

В invoker указывается компонент, относительно которого будет отображаться меню. Значения upperX и upperY определяют позицию X,Y левого верхнего угла меню относительно вызывающего компонента, заданного в invoker. Обычный способ получения вызывающего компонента предусматривает вызов getComponent() на объекте события, переданного обработчику событий мыши.

Изложенное выше можно применить на практике, добавив всплывающее меню Edit (Правка) в программу MenuDemo, созданную в начале главы. Меню Edit будет содержать три пункта: Cut (Вырезать), Copy (Копировать) и Paste (Вставить). Начнем с добавления в класс MenuDemo следующей переменной экземпляра:

```
JPopupMenu jpu;
```

Переменная jpu будет хранить ссылку на всплывающее меню.

Затем нужно добавить в конструктор MenuDemo приведенную ниже кодовую последовательность:

```
// Создать всплывающее меню Edit.
jpu = new JPopupMenu();

// Создать пункты для всплывающего меню.
JMenuItem jmiCut = new JMenuItem("Cut");
JMenuItem jmiCopy = new JMenuItem("Copy");
JMenuItem jmiPaste = new JMenuItem("Paste");

// Добавить пункты во всплывающее меню.
jpu.add(jmiCut);
jpu.add(jmiCopy);
jpu.add(jmiPaste);

// Добавить прослушиватель для инициатора всплывающего меню.
jfrm.addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent me) {
    if(me.isPopupTrigger())
      jpu.show(me.getComponent(), me.getX(), me.getY());
  }
  public void mouseReleased(MouseEvent me) {
    if(me.isPopupTrigger())
      jpu.show(me.getComponent(), me.getX(), me.getY());
  }
});
```

Кодовая последовательность начинается с создания экземпляра JPopupMenu и его сохранения в jpu. Затем обычным образом создаются три пункта меню Cut, Copy и Paste, которые добавляются в jpu. На этом построение всплывающего меню Edit завершено. Всплывающие меню не добавляются ни в панель меню, ни в любой другой объект.

Далее добавляется MouseListener путем создания анонимного внутреннего класса, который основан на классе MouseAdapter, т.е. в прослушивателе придется переопределить только методы, относящиеся к всплывающему меню: mousePressed() и mouseReleased(). Адаптер предоставляет стандартные реализации других методов MouseListener. Обратите внимание,

что прослушиватель событий мыши добавляется в jfrm. Это означает, что щелчок правой кнопкой мыши внутри любой части панели содержимого инициирует отображение всплывающего меню.

Методы mousePressed() и mouseReleased() вызывают isPopupTrigger() для выяснения, оказалось ли событие мыши инициатором всплывающего меню, и если да, то вызовом show() производится отображение всплывающего меню. Вызывающий компонент получается путем вызова getComponent() на объекте события мыши. В данном случае инициатором будет панель содержимого. Координаты X,Y левого верхнего угла получаются посредством вызовов getX() и getY(), что приводит к отображению всплывающего меню с левым верхним углом прямо под указателем мыши.

Наконец, в программу также необходимо добавить показанные ниже прослушиватели событий действий. Они обрабатывают события действий, генерируемые, когда пользователь выбирает элемент во всплывающем меню.

```
jmiCut.addActionListener(this);
jmiCopy.addActionListener(this);
jmiPaste.addActionListener(this);
```

После внесения таких дополнений всплывающее меню можно активизировать, щелкнув правой кнопкой мыши в любом месте панели содержимого приложения. Результат представлен на рис. 34.5.

Рис. 34.5. Всплывающее меню Edit

Еще одно замечание касательно предыдущего примера. Поскольку инициатором всплывающего меню всегда является jfrm, в данном случае его можно передать явно, а не вызывать getComponent(). Для этого jfrm потребуется сделать переменной экземпляра класса MenuDemo (а не локальной переменной), чтобы она была доступной внутреннему классу. Затем для отображения всплывающего меню можно использовать следующий вызов show():

```
jpu.show(jfrm, me.getX(), me.getY());
```

Хотя такой прием работает в рассматриваемом примере, преимущество применения getComponent() связано с тем, что всплывающее меню автоматически отображается относительно вызывающего компонента. Таким образом, один и тот же код можно использовать для отображения любого всплывающего меню относительно вызывающего его объекта.

Создание панели инструментов

Панель инструментов — это компонент, который может служить как альтернативой, так и дополнением к меню. Панель инструментов содержит список кнопок (или других компонентов), которые предоставляют пользователю немедленный доступ к различным параметрам программы. Например, в панели инструментов могут находиться кнопки для выбора различных вариантов написания шрифта, таких как полужирное, курсивное, выделенное или подчеркнутое. Параметры можно выбирать, не обращаясь к меню. Как правило, на кнопках панели инструментов отображаются значки, а не текст, хотя допускается то и другое. Кроме того, с кнопками панели инструментов в виде значков часто связаны всплывающие подсказки. Панели инструментов за счет перетаскивания можно располагать с любой стороны окна, а также перетаскивать их полностью за пределы окна, в результате чего они станут свободно плавающими.

Панели инструментов в Swing являются экземплярами класса `JToolBar`, конструкторы которого позволяют создавать панель инструментов с заголовком или без него. Вдобавок можно указывать компоновку панели инструментов — горизонтальную или вертикальную. Ниже перечислены конструкторы класса `JToolBar`:

```
JToolBar()
JToolBar(String title)
JToolBar(int how)
JToolBar(String title, int how)
```

Первый конструктор создает горизонтальную панель инструментов без заголовка, а второй — горизонтальную панель инструментов с заголовком, указанным в `title`. Заголовок будет отображаться только в случае перетаскивания панели инструментов за пределы окна. Третий конструктор создает панель инструментов, ориентированную в соответствии с параметром `how`, который должен иметь значение либо `JToolBar.VERTICAL`, либо `JToolBar.HORIZONTAL`. Четвертый конструктор создает панель инструментов с заголовком, заданным в `title`, и ориентацией, указанной в `how`.

Панель инструментов обычно применяется с окном, в котором используется граничная компоновка, по двум причинам. Во-первых, это позволяет первоначально расположить панель инструментов вдоль одной из четырех позиций границы, причем часто применяется верхняя позиция. Во-вторых, это позволяет перетаскивать панель инструментов в любую сторону окна.

Помимо перетаскивания панели инструментов в разные места внутри окна ее также можно перетаскивать за пределы окна, что приводит к созданию отстыкованной панели инструментов. Если при создании панели инструментов был указан заголовок, то он будет отображаться, когда панель инструментов окажется отстыкованной. Кнопки (или другие компоненты) добавляются в панель инструментов во многом аналогично их добавлению в панель меню. Нужно просто вызвать метод `add()`. Компоненты отображаются в панели инструментов в порядке их добавления.

Созданная панель инструментов добавляется не в панель меню (даже если она существует), а в контейнер окна. Как уже упоминалось, панель инструментов обычно добавляется в верхнюю (т.е. северную) позицию граничной компоновки с использованием горизонтальной ориентации. Компонент, подвергающийся воздействию, помещается в центральную область граничной компоновки. Применение такого подхода приводит к тому, что после запуска программы панель инструментов оказывается в ожидаемом месте. Однако панель инструментов можно перетащить в любую другую позицию. Разумеется, панель инструментов можно также перетащить за пределы окна.

В целях иллюстрации работы с панелью инструментов она будет добавлена в программу MenuDemo. В панели инструментов будут представлены три параметра отладки: создать точку останова, очистить точку останова и возобновить выполнение программы. Чтобы добавить панель инструментов, понадобится выполнить три шага.

Во-первых, необходимо удалить из программы следующую строку:

```
jfrm.setLayout(new FlowLayout());
```

В результате JFrame автоматически начнет использовать граничную компоновку.

Во-вторых, поскольку применяется BorderLayout, нужно изменить строку, добавляющую метку jlab во фрейм:

```
jfrm.add(jlab, BorderLayout.CENTER);
```

Эта строка явно добавляет jlab в центральную область граничной компоновки. (Формально указывать центральную область явным образом не обязательно, т.к. при использовании граничной компоновки именно туда по умолчанию добавляются компоненты. Тем не менее, явное указание BorderLayout.CENTER дает понять любому читающему код, что применяется граничная компоновка, и метка jlab размещается по центру.)

В-третьих, потребуется добавить приведенный далее код, который создает панель инструментов Debug (Отладка):

```
// Создать панель инструментов Debug.
JToolBar jtb = new JToolBar("Debug");

// Загрузить изображения.
ImageIcon set = new ImageIcon("setBP.gif");
ImageIcon clear = new ImageIcon("clearBP.gif");
ImageIcon resume = new ImageIcon("resume.gif");

// Создать кнопки для панели инструментов.
JButton jbtnSet = new JButton(set);
jbtnSet.setActionCommand("Set Breakpoint");  // Установить точку останова
jbtnSet.setToolTipText("Set Breakpoint");

JButton jbtnClear = new JButton(clear);
jbtnClear.setActionCommand("Clear Breakpoint"); // Очистить точку останова
jbtnClear.setToolTipText("Clear Breakpoint");

JButton jbtnResume = new JButton(resume);
jbtnResume.setActionCommand("Resume");          // Возобновить выполнение
jbtnResume.setToolTipText("Resume");
```

```
// Добавить кнопки в панель инструментов.
jtb.add(jbtnSet);
jtb.add(jbtnClear);
jtb.add(jbtnResume);

// Добавить панель инструментов в северную позицию панели содержимого.
jfrm.add(jtb, BorderLayout.NORTH);
```

Проанализируем код. Сначала создается экземпляр `JToolBar` с заголовком `"Debug"`. Затем создается набор объектов `ImageIcon`, которые содержат изображения для кнопок панели инструментов. Далее создаются три кнопки панели инструментов. Обратите внимание, что у каждой есть изображение, но нет текста. Кроме того, каждой кнопке явно назначены команда действия и всплывающая подсказка. Команды действий устанавливаются из-за того, что кнопкам не назначаются имена при их создании. Всплывающие подсказки особенно полезны, когда применяются к компонентам панели инструментов на основе значков, т.к. временами нелегко создать изображения, интуитивно понятные всем пользователям. Кнопки добавляются в панель инструментов, а панель инструментов — к северной стороне граничной компоновки фрейма.

В заключение добавляются прослушиватели событий действий для панели инструментов:

```
// Добавить прослушиватели событий действий для панели инструментов.
jbtnSet.addActionListener(this);
jbtnClear.addActionListener(this);
jbtnResume.addActionListener(this);
```

Каждый раз, когда пользователь нажимает кнопку в панели инструментов, инициируется событие действия, которое обрабатывается так же, как и другие события, связанные с меню. На рис. 34.6 показана работающая панель инструментов.

Рис. 34.6. Работающая панель инструментов Debug

Использование действий

Зачастую панель инструментов и пункт меню содержат общие элементы. Например, функции панели инструментов Debug из предыдущего примера также могут быть предложены через выбор меню. В таком случае выбор

варианта (скажем, установка точки останова) приводит к выполнению того же самого действия независимо от того, что применялось для выбора — меню или панель инструментов. Кроме того, и кнопка панели инструментов, и пункт меню (вполне вероятно) будут использовать один и тот же значок. Кроме того, когда кнопка панели инструментов отключена, соответствующий пункт меню тоже должен быть отключен. Такая ситуация обычно приводит к большому количеству дублированного взаимозависимого кода, что далеко не оптимально. К счастью, Swing предлагает решение — *действие*.

Действие представляет собой экземпляр реализации интерфейса `Action`, который расширяет интерфейс `ActionListener` и предоставляет средства объединения информации о состоянии с обработчиком событий `actionPerformed()`. Итоговая комбинация позволяет одному действию управлять двумя или более компонентами. Например, действие обеспечивает централизацию управления и обработки для кнопки панели инструментов и пункта меню. Вместо дублирования кода в программе достаточно создать действие, которое автоматически поддерживает оба компонента.

Помимо унаследованного метода `actionPerformed()` в интерфейсе `Action` определено несколько собственных методов. Особый интерес представляет метод `putValue()`, который устанавливает значение различных свойств, связанных с действием:

```
void putValue(String key, Object val)
```

Он присваивает `val` свойству, указанному в `key`. Хотя это не применяется в следующем примере, полезно отметить, что интерфейс `Action` также предлагает метод `getValue()`, который получает значение указанного свойства:

```
Object getValue(String key)
```

Он возвращает ссылку на свойство, заданное в `key`.

В табл. 34.2 кратко описаны значения ключей, используемые методами `putValue()` и `getValue()`.

Таблица 34.2. Значения ключей, используемые методами `putValue()` и `getValue()`

Значение ключа	Описание
`static final String ACCELERATOR_KEY`	Представляет свойство клавиатурного сочетания. Клавиатурные сочетания указываются в виде объектов `KeyStroke`
`static final String ACTION_COMMAND_KEY`	Представляет свойство команды действия. Команда действия указывается в виде строки
`static final String DISPLAYED_MNEMONIC_INDEX_KEY`	Представляет индекс символа, отображаемого как мнемонический. Указывается в виде значения `Integer`

Окончание табл. 34.2

Значение ключа	Описание
`static final String LARGE_ICON_KEY`	Представляет крупный значок, ассоциированный с действием. Значок указывается в виде объекта типа `Icon`
`static final String LONG_DESCRIPTION`	Представляет длинное описание действия. Указывается в виде строки
`static final String MNEMONIC_KEY`	Представляет свойство мнемонического символа. Мнемонический символ указывается в вид константы `KeyEvent`
`static final String NAME`	Представляет имя действия (которое также становится именем кнопки или пункта меню, связанного с действием). Имя указывается в виде строки
`static final String SELECTED_KEY`	Представляет состояние выбора. Если установлено, тогда пункт выбран. Состояние задается с помощью значения `Boolean`
`static final String SHORT_DESCRIPTION`	Представляет текст всплывающей подсказки, ассоциированный с действием. Указывается в виде строки
`static final String SMALL_ICON`	Представляет значок, ассоциированный с действием. Значок указывается в виде объекта типа `Icon`

Например, следующий вызов `putValue()` устанавливает в качестве мнемонического символа букву X:

```
actionOb.putValue(MNEMONIC_KEY, KeyEvent.VK_X);
```

Через методы `putValue()` и `getValue()` недоступно только одно свойство `Action` — состояние "включено/выключено". Для доступа к нему применяются методы `setEnabled()` и `isEnabled()`:

```
void setEnabled(boolean enabled)
boolean isEnabled()
```

Если `enable` равно `true`, то действие включено. В противном случае оно отключено. Если действие разрешено, тогда метод `isEnabled()` возвращает `true`, или `false` в противном случае.

Хотя можно реализовать весь интерфейс `Action` самостоятельно, обычно это не требуется. Взамен Swing предоставляет частичную реализацию по имени `AbstractAction`, которая допускает расширение. При расширении `AbstractAction` понадобится реализовать только один метод:

actionPerformed(). Другие методы интерфейса Action предоставляются автоматически. В классе AbstractAction определены три конструктора, один из которых используется далее в главе:

```
AbstractAction(String name, Icon image)
```

Он создает экземпляр AbstractAction с именем, указанным в name, и значком, заданным в image.

Созданное действие можно добавить в JToolBar и применять при конструировании JMenuItem. Для добавления действия в JToolBar предназначена следующая форма метода add():

```
JButton add(Action actObj)
```

В actObj указывается действие, которое добавляется в панель инструментов. Свойства, определенные в actObj, используются для создания кнопки панели инструментов. Чтобы создать пункт меню из действия, понадобится применить такой конструктор JMenuItem:

```
JMenuItem(Action actObj)
```

Здесь actObj — действие, используемое для создания пункта меню в соответствии с его свойствами.

На заметку! Помимо JToolBar и JMenuItem действия также поддерживаются рядом других компонентов Swing вроде JPopupMenu, JButton, JRadioButton и JCheckBox. Компоненты JRadioButtonMenuItem и JCheckBoxMenuItem тоже поддерживают действия.

Чтобы проиллюстрировать преимущества действий, они будут применяться для управления панелью инструментов Debug, созданной в предыдущем разделе. Кроме того, в главное меню Options также добавляется подменю Debug (Отладка) с теми же пунктами, что и в панели инструментов Debug: Set Breakpoint (Установить точку останова), Clear Breakpoint (Очистить точку останова) и Resume (Возобновить выполнение). Поддерживать эти элементы в меню будут те же действия, которые поддерживают их в панели инструментов. Таким образом, вместо написания повторяющегося кода для обработки панели инструментов и меню оба инструмента будут обрабатываться действиями.

Первым делом необходимо создать внутренний класс по имени DebugAction, который расширяет AbstractAction:

```
// Класс, представляющий действие для меню Debug и панели инструментов.
class DebugAction extends AbstractAction {
  public DebugAction(String name, Icon image, int mnem,
                     int accel, String tTip) {
    super(name, image);
    putValue(ACCELERATOR_KEY,
    KeyStroke.getKeyStroke(accel, InputEvent.CTRL_DOWN_MASK));
    putValue(MNEMONIC_KEY, mnem);
    putValue(SHORT_DESCRIPTION, tTip);
  }
```

```
  // Обработать события для панели инструментов и меню Debug.
  public void actionPerformed(ActionEvent ae) {
    String comStr = ae.getActionCommand();

    jlab.setText(comStr + " Selected");

    // Переключить включенное состояние пунктов
    // Set Breakpoint и Clear Breakpoint.
    if(comStr.equals("Set Breakpoint")) {
      clearAct.setEnabled(true);
      setAct.setEnabled(false);
    } else if(comStr.equals("Clear Breakpoint")) {
      clearAct.setEnabled(false);
      setAct.setEnabled(true);
    }
  }
}
```

Класс `DebugAction` расширяет `AbstractAction` и является классом действия, который будет использоваться для определения свойств, ассоциированных с меню и панелью инструментов Debug. Его конструктор принимает пять параметров, позволяющие указывать следующие элементы:

- имя;
- значок;
- мнемонический символ;
- клавиатурное сочетание;
- всплывающую подсказку.

Первые два свойства передаются конструктору `AbstractAction` через `super`, а оставшиеся три устанавливаются посредством вызовов `putValue()`.

Метод `actionPerformed()` класса `DebugAction` обрабатывает события для действия. Другими словами, когда экземпляр `DebugAction` применяется для создания кнопки панели инструментов и пункта меню, сгенерированные любым из этих компонентов события обрабатываются методом `actionPerformed()` из `DebugAction`. Обратите внимание, что данный обработчик отображает выбор в метке `jlab`. Когда выбрано действие Set Breakpoint, включается пункт Clear Breakpoint и отключается Set Breakpoint, а когда выбрано действие Clear Breakpoint, включается пункт Set Breakpoint и отключается Clear Breakpoint. Именно так действие можно использовать для включения или отключения компонента. Действие отключается для всех своих применений. Например, если действие Set Breakpoint отключено, то оно отключено и в панели инструментов, и в меню.

В класс `MenuDemo` понадобится добавить следующие переменные экземпляра типа `DebugAction`:

```
DebugAction setAct;
DebugAction clearAct;
DebugAction resumeAct;
```

Затем необходимо создать три объекта ImageIcon, представляющие пункты отладки:

```
// Загрузить изображения для действий.
ImageIcon setIcon = new ImageIcon("setBP.gif");
ImageIcon clearIcon = new ImageIcon("clearBP.gif");
ImageIcon resumeIcon = new ImageIcon("resume.gif");
```

Далее нужно создать действия, которые управляют пунктами отладки:

```
// Создать действия.
setAct = new DebugAction("Set Breakpoint",
                         setIcon,
                         KeyEvent.VK_S,
                         KeyEvent.VK_B,
                         "Set a break point.");
clearAct = new DebugAction("Clear Breakpoint",
                         clearIcon,
                         KeyEvent.VK_C,
                         KeyEvent.VK_L,
                         "Clear a break point.");
resumeAct = new DebugAction("Resume",
                         resumeIcon,
                         KeyEvent.VK_R,
                         KeyEvent.VK_R,
                         "Resume execution after breakpoint.");

// Изначально отключить действие Clear Breakpoint.
clearAct.setEnabled(false);
```

Обратите внимание, что клавиатурным сочетанием для Set Breakpoint является B, а клавиатурным сочетанием для Clear Breakpoint — L. Указанные клавиши используются вместо S и C из-за того, что последние уже назначены пунктам Save и Close меню File. Однако их по-прежнему можно применять в качестве мнемонических символов, поскольку каждый мнемонический символ привязан к своему меню. Кроме того, действие, представляющее Clear Breakpoint, изначально отключено. Оно будет включено только после установки точки останова.

Затем действия необходимо использовать для создания кнопок панели инструментов и добавить эти кнопки в панель инструментов:

```
// Создать кнопки панели инструментов, используя действия.
JButton jbtnSet = new JButton(setAct);
JButton jbtnClear = new JButton(clearAct);
JButton jbtnResume = new JButton(resumeAct);

// Создать панель инструментов Debug.
JToolBar jtb = new JToolBar("Breakpoints");

// Добавить кнопки в панель инструментов.
jtb.add(jbtnSet);
jtb.add(jbtnClear);
jtb.add(jbtnResume);

// Добавить панель инструментов в северную позицию панели содержимого.
jfrm.add(jtb, BorderLayout.NORTH);
```

Наконец, понадобится создать меню Debug:

```
// Создать подменю Debug в меню Options панели меню.
// Использовать действия для создания пунктов.
JMenu jmDebug = new JMenu("Debug");
JMenuItem jmiSetBP = new JMenuItem(setAct);
JMenuItem jmiClearBP = new JMenuItem(clearAct);
JMenuItem jmiResume = new JMenuItem(resumeAct);
jmDebug.add(jmiSetBP);
jmDebug.add(jmiClearBP);
jmDebug.add(jmiResume);
jmOptions.add(jmDebug);
```

После внесения таких изменений и дополнений созданные действия будут применяться для управления меню и панелью инструментов Debug. Таким образом, изменение свойства в действии (например, его отключение) повлияет на все случаи использования этого действия. Теперь окно программы будет выглядеть так, как показано на рис. 34.7.

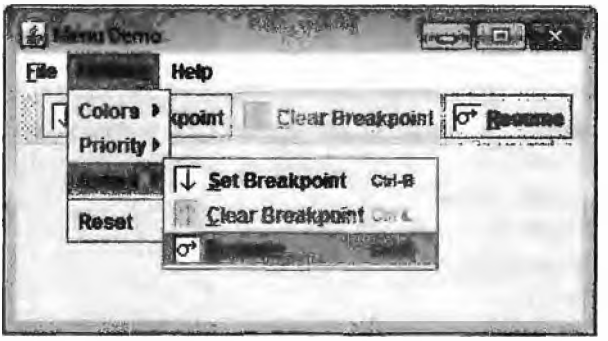


Рис. 34.7. Использование действий для управления панелью инструментов и меню Debug

Построение окончательной программы MenuDemo

В ходе обсуждения в программу MenuDemo, показанную в начале главы, было внесено много изменений и дополнений. Прежде чем закончить, полезно собрать все части вместе, что не только устранит любую двусмысленность в отношении того, как элементы сочетаются друг с другом, но и предоставит полноценную программу, демонстрирующую работу меню, с которой можно проводить эксперименты.

Представленная ниже версия MenuDemo включает в себя все дополнения и усовершенствования, описанные в настоящей главе. Ради ясности программа была реорганизована с применением отдельных методов для создания различных меню и панелей инструментов. Обратите внимание, что некоторые переменные, относящиеся к меню, такие как jmb, jmFile и jtb, были преобразованы в переменные экземпляра.

```
// Полная программа MenuDemo.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MenuDemo implements ActionListener {
  JLabel jlab;
  JMenuBar jmb;
  JToolBar jtb;

  JPopupMenu jpu;

  DebugAction setAct;
  DebugAction clearAct;
  DebugAction resumeAct;

  MenuDemo() {
    // Создать контейнер JFrame.
    JFrame jfrm = new JFrame("Complete Menu Demo");

    // Использовать стандартную граничную компоновку.

    // Установить начальные размеры фрейма.
    jfrm.setSize(360, 200);

    // Завершить работу, когда пользователь закрывает приложение.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создать метку, которая будет отображать выбор меню.
    jlab = new JLabel();

    // Создать панель меню.
    jmb = new JMenuBar();

    // Создать меню File.
    makeFileMenu();

    // Создать действия отладки.
    makeActions();

    // Создать панель инструментов.
    makeToolBar();

    // Создать меню Options.
    makeOptionsMenu();

    // Создать меню Help.
    makeHelpMenu();

    // Создать всплывающее меню Edit.
    makeEditPUMenu();

    // Добавить прослушиватель для инициатора всплывающего меню.
    jfrm.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        if(me.isPopupTrigger())
          jpu.show(me.getComponent(), me.getX(), me.getY());
        }
      public void mouseReleased(MouseEvent me) {
        if(me.isPopupTrigger())
          jpu.show(me.getComponent(), me.getX(), me.getY());
      }
    });
```

```
  // Добавить метку в центральную область панели содержимого.
  jfrm.add(jlab, SwingConstants.CENTER);

  // Добавить панель инструментов в северную область панели содержимого.
  jfrm.add(jtb, BorderLayout.NORTH);

  // Добавить панель меню во фрейм.
  jfrm.setJMenuBar(jmb);

  // Отобразить фрейм.
  jfrm.setVisible(true);
}

// Обработать события действий для пунктов меню.
// События, генерируемые пунктами Debug, НЕ обрабатываются.
public void actionPerformed(ActionEvent ae) {
  // Получить команду действия из выбора меню.
  String comStr = ae.getActionCommand();

  // Если пользователь выбрал Exit, тогда завершить программу.
  if(comStr.equals("Exit")) System.exit(0);

  // Иначе отобразить выбор.
  jlab.setText(comStr + " Selected");
}

// Класс, представляющий действие для меню Debug и панели инструментов.
class DebugAction extends AbstractAction {
  public DebugAction(String name, Icon image, int mnem,
                     int accel, String tTip) {
    super(name, image);
    putValue(ACCELERATOR_KEY,
             KeyStroke.getKeyStroke(accel,
                                    InputEvent.CTRL_DOWN_MASK));
    putValue(MNEMONIC_KEY, mnem);
    putValue(SHORT_DESCRIPTION, tTip);
  }

  // Обработать события для панели инструментов и меню Debug.
  public void actionPerformed(ActionEvent ae) {
    String comStr = ae.getActionCommand();

    jlab.setText(comStr + " Selected");

    // Переключить включенное состояние пунктов
    // Set Breakpoint и Clear Breakpoint.
    if(comStr.equals("Set Breakpoint")) {
      clearAct.setEnabled(true);
      setAct.setEnabled(false);
    } else if(comStr.equals("Clear Breakpoint")) {
      clearAct.setEnabled(false);
      setAct.setEnabled(true);
    }
  }
}

// Создать меню File с мнемоническими символами и клавиатурными сочетаниями
void makeFileMenu() {
  JMenu jmFile = new JMenu("File");
  jmFile.setMnemonic(KeyEvent.VK_F);
```

```
  JMenuItem jmiOpen = new JMenuItem("Open",
                                    KeyEvent.VK_O);
  jmiOpen.setAccelerator(
          KeyStroke.getKeyStroke(KeyEvent.VK_O,
                                 InputEvent.CTRL_DOWN_MASK));
  JMenuItem jmiClose = new JMenuItem("Close",
                                     KeyEvent.VK_C);
  jmiClose.setAccelerator(
           KeyStroke.getKeyStroke(KeyEvent.VK_C,
                                  InputEvent.CTRL_DOWN_MASK));
  JMenuItem jmiSave = new JMenuItem("Save",
                                    KeyEvent.VK_S);
  jmiSave.setAccelerator(
          KeyStroke.getKeyStroke(KeyEvent.VK_S,
                                 InputEvent.CTRL_DOWN_MASK));
  JMenuItem jmiExit = new JMenuItem("Exit",
                                    KeyEvent.VK_E);
  jmiExit.setAccelerator(
          KeyStroke.getKeyStroke(KeyEvent.VK_E,
                                 InputEvent.CTRL_DOWN_MASK));
  jmFile.add(jmiOpen);
  jmFile.add(jmiClose);
  jmFile.add(jmiSave);
  jmFile.addSeparator();
  jmFile.add(jmiExit);
  jmb.add(jmFile);
  // Добавить прослушиватели событий действий для меню File.
  jmiOpen.addActionListener(this);
  jmiClose.addActionListener(this);
  jmiSave.addActionListener(this);
  jmiExit.addActionListener(this);
}
// Создать меню Options.
void makeOptionsMenu() {
  JMenu jmOptions = new JMenu("Options");
  // Создать подменю Colors.
  JMenu jmColors = new JMenu("Colors");
  // Использовать флажки для цветов, что позволит
  // пользователю выбирать более одного цвета.
  JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");
  JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
  JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");
  // Добавить пункты в меню Colors.
  jmColors.add(jmiRed);
  jmColors.add(jmiGreen);
  jmColors.add(jmiBlue);
  jmOptions.add(jmColors);
  // Создать подменю Priority.
  JMenu jmPriority = new JMenu("Priority");
```

```
    // Использовать переключатели для настройки приоритета. Они дают
    // возможность показывать в меню применяемый приоритет и также
    // гарантируют, что в любой момент времени может быть выбран один
    // и только один приоритет. Обратите внимание, что изначально
    // выбран переключатель High.
    JRadioButtonMenuItem jmiHigh =
      new JRadioButtonMenuItem("High", true);
    JRadioButtonMenuItem jmiLow =
      new JRadioButtonMenuItem("Low");

    // Добавить пункты в меню Priority.
    jmPriority.add(jmiHigh);
    jmPriority.add(jmiLow);
    jmOptions.add(jmPriority);

    // Создать группу кнопок для пунктов меню с переключателями.
    ButtonGroup bg = new ButtonGroup();
    bg.add(jmiHigh);
    bg.add(jmiLow);

    // Создать подменю Debug в меню Options панели меню.
    // Использовать действия для создания пунктов.
    JMenu jmDebug = new JMenu("Debug");
    JMenuItem jmiSetBP = new JMenuItem(setAct);
    JMenuItem jmiClearBP = new JMenuItem(clearAct);
    JMenuItem jmiResume = new JMenuItem(resumeAct);

    // Добавить пункты в меню Debug.
    jmDebug.add(jmiSetBP);
    jmDebug.add(jmiClearBP);
    jmDebug.add(jmiResume);
    jmOptions.add(jmDebug);

    // Создать пункт меню Reset.
    JMenuItem jmiReset = new JMenuItem("Reset");
    jmOptions.addSeparator();
    jmOptions.add(jmiReset);

    // Добавить все меню Options в панель меню.
    jmb.add(jmOptions);

    // Добавить прослушиватели событий действий для меню Options
    // за исключением тех, которые поддерживаются меню Debug.
    jmiRed.addActionListener(this);
    jmiGreen.addActionListener(this);
    jmiBlue.addActionListener(this);
    jmiHigh.addActionListener(this);
    jmiLow.addActionListener(this);
    jmiReset.addActionListener(this);
  }

  // Создать меню Help.
  void makeHelpMenu() {
    JMenu jmHelp = new JMenu("Help");

    // Добавить значок к пункту меню About.
    ImageIcon icon = new ImageIcon("AboutIcon.gif");
```

```
    JMenuItem jmiAbout = new JMenuItem("About", icon);
    jmiAbout.setToolTipText("Info about the MenuDemo program.");
    jmHelp.add(jmiAbout);
    jmb.add(jmHelp);

    // Добавить прослушиватель событий действий для пункта меню About.
    jmiAbout.addActionListener(this);
  }

  // Создать действия, необходимые для панели инструментов и меню Debug.
  void makeActions() {
    // Загрузить изображения для действий.
    ImageIcon setIcon = new ImageIcon("setBP.gif");
    ImageIcon clearIcon = new ImageIcon("clearBP.gif");
    ImageIcon resumeIcon = new ImageIcon("resume.gif");

    // Создать действия.
    setAct =
      new DebugAction("Set Breakpoint",
                      setIcon,
                      KeyEvent.VK_S,
                      KeyEvent.VK_B,
                      "Set a break point.");

    clearAct =
      new DebugAction("Clear Breakpoint",
                      clearIcon,
                      KeyEvent.VK_C,
                      KeyEvent.VK_L,
                      "Clear a break point.");

    resumeAct =
      new DebugAction("Resume",
                      resumeIcon,
                      KeyEvent.VK_R,
                      KeyEvent.VK_R,
                      "Resume execution after breakpoint.");

    // Изначально отключить действие Clear Breakpoint.
    clearAct.setEnabled(false);
  }

  // Создать панель инструментов Debug.
  void makeToolBar() {
    // Создать кнопки панели инструментов с использованием действий.
    JButton jbtnSet = new JButton(setAct);
    JButton jbtnClear = new JButton(clearAct);
    JButton jbtnResume = new JButton(resumeAct);

    // Создать панель инструментов Debug.
    jtb = new JToolBar("Breakpoints");

    // Добавить кнопки в панель инструментов.
    jtb.add(jbtnSet);
    jtb.add(jbtnClear);
    jtb.add(jbtnResume);
  }
```

```
  // Создать всплывающее меню Edit.
  void makeEditPUMenu() {
    jpu = new JPopupMenu();

    // Создать пункты всплывающего меню.
    JMenuItem jmiCut = new JMenuItem("Cut");
    JMenuItem jmiCopy = new JMenuItem("Copy");
    JMenuItem jmiPaste = new JMenuItem("Paste");

    // Добавить пункты во всплывающее меню.
    jpu.add(jmiCut);
    jpu.add(jmiCopy);
    jpu.add(jmiPaste);

    // Добавить прослушиватели событий действий для всплывающего меню Edit
    jmiCut.addActionListener(this);
    jmiCopy.addActionListener(this);
    jmiPaste.addActionListener(this);
  }
  public static void main(String[] args) {
    // Создать фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new MenuDemo();
      }
    });
  }
}
```

Продолжение исследования Swing

В инфраструктуре Swing определен очень большой инструментальный набор для построения графических пользовательских инструментов. Существует много других функциональных средств, которые рекомендуется исследовать самостоятельно. Например, есть классы диалоговых окон, такие как `JOptionPane` и `JDialog`, которые можно использовать для упрощения создания диалоговых окон. Кроме того, предлагаются дополнительные элементы управления помимо тех, которые были представлены в главе 33. Двумя из них, которые стоит изучить, являются `JSpinner` (создающий элемент управления типа счетчика) и `JFormattedTextField` (поддерживающий форматированный текст). Также имеет смысл попытаться определить собственные модели для разнообразных компонентов. Откровенно говоря, лучший способ ознакомиться с возможностями инфраструктуры Swing — как следует поэкспериментировать с ней.

ЧАСТЬ

IV

Применение Java

ГЛАВА

35 Архитектура JavaBeans

В главе представлен обзор создания компонентов Java Beans, или Bean-компонентов Java. Они важны, поскольку позволяют создавать сложные системы из программных компонентов. Bean-компоненты Java могут быть предоставлены вами или одним либо несколькими поставщиками. Они применяют архитектуру под названием *JavaBeans*, которая определяет, каким образом такие стандартные блоки могут работать вместе.

Чтобы лучше понять ценность Bean-компонентов, обратимся к аналогии. Разработчики оборудования располагают широким спектром компонентов, которые можно интегрировать вместе для создания системы. Примерами простых строительных блоков можно считать резисторы, конденсаторы и катушки индуктивности. Интегральные схемы обеспечивают более развитую функциональность. Все имеющиеся средства могут использоваться многократно. Нет никакой необходимости или возможности перестраивать их каждый раз, когда требуется новая система. Кроме того, одни и те же детали могут применяться в схемах различных типов. Все становится возможным из-за того, что поведение таких компонентов хорошо понятно и документировано.

В индустрии программного обеспечения тоже велись поиски способов извлечения преимуществ из возможностей многократного использования и взаимодействия подхода на основе компонентов. Для этого необходима компонентная архитектура, которая позволяет собирать программы из программных строительных блоков поставляемых даже разными поставщиками. Разработчик также должен быть в состоянии выбрать компонент, понять предлагаемые им возможности и встроить его в приложение. Когда становится доступной новая версия компонента, включение такой функциональности в существующий код должно делаться легко. Именно такой подход внедрен в архитектуре JavaBeans.

Что собой представляет Bean-компонент

Bean-компонент Java представляет собой программный компонент, предназначенный для многократного использования в различных средах. Какие-либо ограничения на возможности Bean-компонента отсутствуют. Он может

выполнять простую функцию, скажем, получать стоимость складских запасов, или же сложную функцию, например, прогнозировать показатели портфеля акций. Bean-компонент может быть видимым для конечного пользователя. Примером может служить кнопка в графическом пользовательском интерфейсе. Bean-компонент также может быть невидимым для пользователя. Примером строительного блока такого типа является программное обеспечение для декодирования потока мультимедиа-информации в реальном времени. Наконец, Bean-компонент может быть спроектирован для автономной работы на рабочей станции пользователя или для совместной работы с набором других распределенных компонентов. Программное обеспечение для создания круговой диаграммы из набора точек данных является примером Bean-компонента, который может выполняться локально. Однако Bean-компонент, который предоставляет информацию о ценах в режиме реального времени с фондовой или товарной биржи, для получения своих данных должен взаимодействовать с другим распределенным программным обеспечением.

Преимущества Bean-компонентов

Ниже перечислены преимущества, которые технология JavaBeans предоставляет разработчику компонентов.

- Bean-компонент получает все преимущества принятой в Java парадигмы "написанное однажды выполняется везде, в любое время, всегда".
- Свойства, события и методы Bean-компонента, доступные в другом приложении, поддаются контролю.
- Для помощи в настройке Bean-компонента можно применять вспомогательное программное обеспечение, которое необходимо лишь при установке параметров на стадии разработки компонента. Его не нужно включать в среду выполнения.
- Состояние Bean-компонента можно сохранить в постоянном хранилище и позже восстановить.
- Bean-компонент может регистрироваться для получения событий от других объектов и генерировать события, которые отправляются другим объектам.

Самоанализ

В основе программирования Bean-компонентов лежит *самоанализ* — процесс анализа Bean-компонента с целью выяснения его возможностей. Это важная функция JavaBeans API, поскольку она позволяет другому приложению, например, инструменту проектирования, получать информацию о компоненте. Без самоанализа технология JavaBeans не смогла бы работать.

Разработчик компонента может указать, какие из его свойств, событий и методов должны быть доступны, двумя способами. Первый способ предусма-

тривает использование простых соглашений об именовании, которые позволяют механизмам самоанализа выводить информацию о Bean-компоненте. Второй способ связан с созданием дополнительного класса, расширяющего интерфейс BeanInfo, который явно предоставляет такую информацию. Ниже рассматриваются оба способа.

Паттерны проектирования для свойств

Свойство является подмножеством состояния Bean-компонента. Значения, присвоенные свойствам, определяют поведение и внешний вид компонента. Свойство устанавливается с помощью метода *установки*, а получается методом *получения*. Существуют два типа свойств: простые и индексированные.

Простые свойства

Простое свойство имеет одиночное значение. Его можно идентифицировать по следующим паттернам проектирования, где N — имя свойства, а T — его тип:

```
public T getN()
public void setN(T arg)
```

Свойство, допускающее чтение и запись, для доступа к своему значению имеет оба указанных метода. Свойство только для чтения располагает только методом получения, а свойство только для записи — только методом установки.

Вот три простых свойства для чтения и записи вместе с их методами получения и установки:

```
private double depth, height, width;

public double getDepth() {
  return depth;
}

public void setDepth(double d) {
  depth = d;
}

public double getHeight() {
  return height;
}

public void setHeight(double h) {
  height = h;
}

public double getWidth() {
  return width;
}

public void setWidth(double w) {
  width = w;
}
```

На заметку! Для доступа к булевскому свойству можно также применять метод вида `is`*`ИмяСвойства`*`()`.

Индексированные свойства

Индексированное свойство состоит из нескольких значений. Его можно идентифицировать по приведенным далее паттернам проектирования, где `N` — имя свойства, а `T` — его тип:

```
public T getN(int index);
public void setN(int index, T value);
public T[] getN();
public void setN(T[] values);
```

Ниже показано индексированное свойство по имени `data` вместе с его методами получения и установки:

```
private double[] data;

public double getData(int index) {
  return data[index];
}
public void setData(int index, double value) {
  data[index] = value;
}
public double[] getData() {
  return data;
}
public void setData(double[] values) {
  data = new double[values.length];
  System.arraycopy(values, 0, data, 0, values.length);
}
```

Паттерны проектирования для событий

Bean-компоненты используют модель делегирования обработки событий, которая обсуждалась ранее в книге. Компоненты могут генерировать события и отправлять их другим объектам. Их можно идентифицировать по следующим паттернам проектирования, где `T` — тип события:

```
public void addTListener(TListener eventListener)
public void addTListener(TListener eventListener)
  throws java.util.TooManyListenersException
public void removeTListener(TListener eventListener)
```

Перечисленные выше методы применяются для добавления и удаления прослушивателя указанного события. Версия `addTListener()`, которая не генерирует исключение, может использоваться для *групповой рассылки* события, т.е. для получения уведомлений о событии могут зарегистрироваться несколько прослушивателей. Версия, генерирующая исключение `TooManyListenersException`, выполняет *индивидуальную рассылку* события, т.е. количество прослушивателей ограничивается одним. В любом случае для удаления прослушивателя применяется метод `removeTListener()`.

Например, при наличии интерфейса прослушивателя событий по имени TemperatureListener в Bean-компоненте для мониторинга температуры могут быть предоставлены следующие методы:

```
public void addTemperatureListener(TemperatureListener tl) {
  ...
}
public void removeTemperatureListener(TemperatureListener tl) {
  ...
}
```

Методы и паттерны проектирования

Паттерны проектирования не используются при именовании методов, не относящихся к свойствам. Механизм самоанализа находит все открытые методы Bean-компонента. Защищенные и закрытые методы остаются недоступными.

Использование интерфейса BeanInfo

Как показывает предыдущее обсуждение, паттерны проектирования *неявно* определяют, какая информация доступна пользователю Bean-компонента. Интерфейс BeanInfo позволяет *явно* управлять тем, какая информация доступна. В нем определено несколько методов, в том числе:

```
PropertyDescriptor[] getPropertyDescriptors()
EventSetDescriptor[] getEventSetDescriptors()
MethodDescriptor[] getMethodDescriptors()
```

Они возвращают массивы объектов, которые предоставляют информацию о свойствах, событиях и методах Bean-компонента. Классы PropertyDescriptor, EventSetDescriptor и MethodDescriptor находятся в пакете java.beans и описывают указанные элементы. Реализуя эти методы, разработчик может точно определить, что доступно пользователю, пропуская самоанализ, который базируется на паттернах проектирования.

Создаваемый класс, реализующий интерфейс BeanInfo, должен быть назван bnameBeanInfo, где bname — имя Bean-компонента. Например, если Bean-компонент называется MyBean, то классу информации о нем понадобится назначить имя MyBeanBeanInfo.

Для упрощения работа с BeanInfo в JavaBeans предлагается класс SimpleBeanInfo, который предоставляет стандартную реализацию интерфейса BeanInfo, включая три только что показанных метода. Класс SimpleBeanInfo можно расширить и переопределить один или несколько методов, чтобы явно управлять тем, какие аспекты Bean-компонента будут доступны. Если какой-либо метод не переопределен, тогда будет применяться самоанализ на основе паттернов проектирования. Например, если не был переопределен метод getPropertyDescriptors(), то для обнаружения свойств Bean-компонента используются паттерны проектирования. Позже в главе вы увидите класс SimpleBeanInfo в действии.

Связанные и ограниченные свойства

Bean-компонент, имеющий *связанное* свойство, при изменении этого свойства генерирует событие типа `PropertyChangeEvent`, которое отправляется объектам, ранее зарегистрировавшим свой интерес в получении таких уведомлений. Класс, обрабатывающий событие, должен реализовывать интерфейс `PropertyChangeListener`.

Bean-компонент с *ограниченным* свойством генерирует событие при попытке изменить его значение. Он также генерирует событие типа `PropertyChangeEvent`, которое тоже рассылается объектам, ранее заявившим о своей заинтересованности в получении уведомлений подобного рода. Тем не менее, другие объекты имеют возможность наложить запрет на предложенное изменение, сгенерировав исключение `PropertyVetoException`. Такая возможность позволяет Bean-компоненту работать по-разному в зависимости от среды выполнения. Класс, обрабатывающий событие `PropertyVetoException`, должен реализовывать интерфейс `VetoableChangeListener`.

Постоянство

Постоянство — это способность сохранять текущее состояние Bean-компонента, включая значения его свойств и переменных экземпляра, в энергонезависимой памяти и извлекать их в более позднее время. Для обеспечения постоянства Bean-компонентов применяются возможности сериализации объектов, предоставляемые библиотеками классов Java.

Самый простой способ сериализировать Bean-компонент предусматривает реализацию интерфейса `java.io.Serializable`, который является просто маркерным интерфейсом. Реализация `java.io.Serializable` делает сериализацию автоматической. Bean-компоненту не придется предпринимать какие-то другие действия. Автоматическая сериализация также может наследоваться, т.е. если какой-либо суперкласс Bean-компонента реализует интерфейс `java.io.Serializable`, то будет обеспечена автоматическая сериализация.

При использовании автоматической сериализации можно предотвратить сохранение поля с помощью ключевого слова `transient`. Таким образом, элементы данных Bean-компонента, указанные как `transient`, сериализироваться не будут.

Если Bean-компонент не реализует интерфейс `java.io.Serializable`, тогда придется обеспечить сериализацию самостоятельно, например, за счет реализации интерфейса `java.io.Externalizable`, иначе контейнеры не смогут сохранять конфигурацию Bean-компонента.

Настройщики

Разработчик Bean-компонента может предоставить *настройщик*, который помогает другому разработчику конфигурировать Bean-компонент.

Настройщик может обеспечивать пошаговое руководство по процессу, которому необходимо следовать, чтобы задействовать компонент в определенном контексте. Также может быть предложена онлайновая документация. В распоряжении разработчика Bean-компонента доступно достаточно средств для создания настройщика, который позволит представить его продукт на рынке в выгодном свете.

JavaBeans API

Функциональность JavaBeans обеспечивается набором классов и интерфейсов из пакета `java.beans`. Начиная с версии JDK 9, данный пакет находится в модуле `java.desktop`. В этом разделе представлен краткий обзор его содержимого. В табл. 35.1 перечислены интерфейсы в `java.beans`, которые не объявлены нерекомендуемыми, и приведено краткое описание их функциональных возможностей. В табл. 35.2 перечислены классы в `java.beans`.

Таблица 35.1. Интерфейсы в `java.beans`, не объявленные нерекомендуемыми

Интерфейс	Описание
`BeanInfo`	Этот интерфейс позволяет разработчику указать информацию о свойствах, событиях и методах Bean-компонента
`Customizer`	Этот интерфейс позволяет разработчику предоставить графический пользовательский интерфейс для конфигурирования Bean-компонента
`DesignMode`	Методы в этом интерфейсе определяют, функционирует ли Bean-компонент в режиме проектирования
`ExceptionListener`	Методы в этом интерфейсе вызываются при возникновении исключения
`PropertyChangeListener`	Методы в этом интерфейсе вызываются при изменении связанного свойства
`PropertyEditor`	Объекты, реализующие этот интерфейс, позволяют разработчику изменять и отображать значения свойств
`VetoableChangeListener`	Методы в этом интерфейсе вызываются при изменении ограниченного свойства
`Visibility`	Методы в этом интерфейсе позволяют Bean-компоненту функционировать в средах, где графический пользовательский интерфейс недоступен

Таблица 35.2. Классы в java.beans

Класс	Описание
BeanDescriptor	Предоставляет информацию о Bean-компоненте. Также позволяет ассоциировать настройщик с Bean-компонентом
Beans	Используется для получения информации о Bean-компоненте
DefaultPersistenceDelegate	Конкретный подкласс класса PersistenceDelegate
Encoder	Кодирует состояние набора Bean-компонентов. Может применяться для записывания этой информации в поток данных
EventHandler	Поддерживает динамическое создание прослушивателей событий
EventSetDescriptor	Экземпляры этого класса описывают событие, которое может быть сгенерировано Bean-компонентом
Expression	Инкапсулирует вызов метода, который возвращает результат
FeatureDescriptor	Суперкласс классов PropertyDescriptor, EventSetDescriptor и MethodDescriptor (помимо прочих)
IndexedPropertyChangeEvent	Подкласс класса PropertyChangeEvent, который представляет изменение, внесенное в индексированное свойство
IndexedPropertyDescriptor	Экземпляры этого класса описывают индексированное свойство Bean-компонента
IntrospectionException	Исключение этого типа генерируется в случае возникновения проблемы при анализе Bean-компонента
Introspector	Этот класс анализирует Bean-компонент и конструирует объект BeanInfo, который описывает данный компонент
MethodDescriptor	Экземпляры этого класса описывают методы Bean-компонента
ParameterDescriptor	Экземпляры этого класса описывают параметры методов Bean-компонента

Окончание табл. 35.2

Класс	Описание
PersistenceDelegate	Обрабатывает информацию о состоянии объекта
PropertyChangeEvent	Это событие генерируется при изменении связанных или ограниченных свойств. Оно отправляется объектам, которые зарегистрировали интерес к таким событиям и реализуют интерфейс прослушивателя PropertyChangeListener или VetoableChangeListener
PropertyChangeListenerProxy	Расширяет EventListenerProxy и реализует PropertyChangeListener
PropertyChangeSupport	Bean-компоненты, которые поддерживают связанные свойства, могут использовать этот класс для уведомления объектов реализации PropertyChangeListener
PropertyDescriptor	Экземпляры этого класса описывают свойства Bean-компонента
PropertyEditorManager	Этот класс находит объект PropertyEditor для данного типа
PropertyEditorSupport	Этот класс обеспечивает функциональность, которую можно использовать при написании редакторов свойств
PropertyVetoException	Исключение этого типа генерируется, если внесение изменения в ограниченное свойство запрещено
SimpleBeanInfo	Этот класс обеспечивает функциональность, которую можно использовать при написании классов BeanInfo
Statement	Инкапсулирует вызов метода
VetoableChangeListenerProxy	Расширяет EventListenerProxy и реализует VetoableChangeListener
VetoableChangeSupport	Bean-компоненты, которые поддерживают ограниченные свойства, могут использовать этот класс для уведомления объектов реализации VetoableChangeListener
XMLDecoder	Применяется для чтения Bean-компонента из документа XML
XMLEncoder	Используется для записи Bean-компонента в документ XML

Хотя обсуждение всех классов выходит за рамки настоящей главы, особый интерес представляют четыре класса: Introspector, PropertyDescriptor, EventSetDescriptor и MethodDescriptor. Все они кратко рассматриваются далее в главе.

Introspector

Класс Introspector предоставляет несколько статических методов, поддерживающих самоанализ. Наиболее интересный из них — метод getBeanInfo(), возвращающий объект BeanInfo, который можно применять для получения информации о Bean-компоненте. Метод getBeanInfo() имеет несколько форм, в том числе показанную ниже:

```
static BeanInfo getBeanInfo(Class<?> bean) throws IntrospectionException
```

Возвращаемый объект содержит информацию о Bean-компоненте, указанном в bean.

PropertyDescriptor

Класс PropertyDescriptor описывает характеристики свойства Bean-компонента. Он поддерживает несколько методов для управления и описания свойств. Скажем, вызов метода isBound() позволяет выяснить, является свойство связанным или нет, а вызов метода isConstrained() — определить, ограничено свойство или нет. Чтобы получить имя свойства, необходимо вызвать метод getName().

EventSetDescriptor

Класс EventSetDescriptor представляет набор событий Bean-компонента. В нем поддерживается несколько методов, которые получают методы, используемые Bean-компонентом для добавления или удаления прослушивателей событий, а также для других способов управления событиями. Например, чтобы получить метод, применяемый для добавления прослушивателей, понадобится вызвать getAddListenerMethod(), а чтобы получить метод, используемый для удаления прослушивателей, необходимо вызвать getRemoveListenerMethod(). Для получения типа прослушивателя нужно вызвать getListenerType(). Получить имя набора событий можно с помощью метода getName().

MethodDescriptor

Класс MethodDescriptor представляет метод Bean-компонента. Для получения имени метода необходимо вызвать getName(). Для получения информации о методе понадобится вызвать getMethod():

```
Method getMethod()
```

Возвращается объект типа Method, который описывает метод.

Пример Bean-компонента

В заключение главы приводится пример, который иллюстрирует различные аспекты программирования Bean-компонентов, включая самоанализ и применение класса BeanInfo. В нем также используются классы Introspector, PropertyDescriptor и EventSetDescriptor. В примере определены три класса. Первый из них — Bean-компонент по имени Colors:

```
// Простой Bean-компонент.
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

public class Colors extends Canvas implements Serializable {
  transient private Color color;          // не сохраняется
  private boolean rectangular;            // сохраняется

  public Colors() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        change();
      }
    });
    rectangular = false;
    setSize(200, 100);
    change();
  }

  public boolean getRectangular() {
    return rectangular;
  }

  public void setRectangular(boolean flag) {
    this.rectangular = flag;
    repaint();
  }

  public void change() {
    color = randomColor();
    repaint();
  }

  private Color randomColor() {
    int r = (int)(255*Math.random());
    int g = (int)(255*Math.random());
    int b = (int)(255*Math.random());
    return new Color(r, g, b);
  }

  public void paint(Graphics g) {
    Dimension d = getSize();
    int h = d.height;
    int w = d.width;
    g.setColor(color);
    if(rectangular) {
      g.fillRect(0, 0, w-1, h-1);
    }
```

```
    else {
      g.fillOval(0, 0, w-1, h-1);
    }
  }
}
```

Bean-компонент `Colors` отображает раскрашенный объект внутри фрейма. Цвет компонента определяется закрытой переменной `color` типа `Color`, а его форма — закрытой булевской переменной `rectangular`. В конструкторе определен анонимный внутренний класс, который расширяет `MouseAdapter` и переопределяет его метод `mousePressed()`. В ответ на нажатие кнопки мыши вызывается метод `change()`, который выбирает случайный цвет и затем перерисовывает компонент. Методы `getRectangular()` и `setRectangular()` обеспечивают доступ к одному свойству этого Bean-компонента. В методе `change()` вызывается `randomColor()` для выбора цвета, а затем `repaint()`, чтобы изменение стало видимым. Обратите внимание, что в методе `paint()` для определения, как представить Bean-компонент, применяются переменные `rectangular` и `color`.

Вторым классом является `ColorsBeanInfo` — подкласс `SimpleBeanInfo`, предоставляющий явную информацию о `Colors`.

Метод `getPropertyDescriptors()` в нем переопределен, чтобы указывать, какие свойства будут доступны пользователю Bean-компонента. В данном случае единственное доступное свойство — `rectangular`. Метод создает и возвращает объект `PropertyDescriptor` для свойства `rectangular`. Вот используемый конструктор `PropertyDescriptor`:

```
PropertyDescriptor(String property, Class<?> beanCls)
  throws IntrospectionException
```

В первом аргументе указывается имя свойства, а во втором — класс Bean-компонента.

```
// Класс информации о Bean-компоненте.

import java.beans.*;

public class ColorsBeanInfo extends SimpleBeanInfo {
  public PropertyDescriptor[] getPropertyDescriptors() {
    try {
      PropertyDescriptor rectangular =
        new PropertyDescriptor("rectangular", Colors.class);
      PropertyDescriptor[] pd = {rectangular};
      return pd;
    }
    catch(Exception e) {
      System.out.println("Exception caught: " + e); // Возникло исключение
    }
    return null;
  }
}
```

Третий класс называется `IntrospectorDemo`. Он применяет самоанализ для отображения свойств и событий, доступных в Bean-компоненте `Colors`.

```
// Отображение свойств и событий.
import java.awt.*;
import java.beans.*;
public class IntrospectorDemo {
  public static void main(String[] args) {
    try {
      Class<?> c = Class.forName("Colors");
      BeanInfo beanInfo = Introspector.getBeanInfo(c);
      System.out.println("Свойства:");
      PropertyDescriptor[] propertyDescriptor =
        beanInfo.getPropertyDescriptors();
      for(int i = 0; i < propertyDescriptor.length; i++) {
        System.out.println("\t" + propertyDescriptor[i].getName());
      }
      System.out.println("События:");
      EventSetDescriptor[] eventSetDescriptor =
        beanInfo.getEventSetDescriptors();
      for(int i = 0; i < eventSetDescriptor.length; i++) {
        System.out.println("\t" + eventSetDescriptor[i].getName());
      }
    }
    catch(Exception e) {
      System.out.println("Exception caught: " + e); // Возникло исключение
    }
  }
}
```

Вывод программы выглядит следующим образом:

```
Свойства:
      rectangular
События:
      mouseWheel
      mouse
      mouseMotion
      component
      hierarchyBounds
      focus
      hierarchy
      propertyChange
      inputMethod
      key
```

Обратите внимание на два момента в выводе. Во-первых, поскольку метод `getPropertyDescriptors()` в классе `ColorsBeanInfo` переопределен так, что возвращается единственное свойство `rectangular`, то только оно и отображается. Однако из-за того, что метод `getEventSetDescriptors()` не был переопределен в `ColorsBeanInfo`, используется самоанализ на основе паттернов проектирования, и обнаруживаются все события, в том числе из родительского класса `Canvas`. Не забывайте, что если не переопределить один из методов получения в `SimpleBeanInfo`, то по умолчанию применяется самоанализ на базе паттернов проектирования. Чтобы увидеть разницу, которую привносит `ColorsBeanInfo`, удалите его файл класса и снова запустите `IntrospectorDemo`. На этот раз отобразится большее количество свойств.

ГЛАВА

36 Введение в сервлеты

В настоящей главе предлагается введение в *сервлеты*. Сервлеты — это небольшие программы, которые выполняются на серверной стороне веб-подключения. Тема сервлетов довольно обширна и ее полное раскрытие выходит за рамки главы. Взамен основное внимание уделяется ключевым концепциям, интерфейсам и классам, после чего рассматривается несколько примеров.

Происхождение сервлетов

Для понимания преимуществ сервлетов необходимо иметь общее представление о том, как веб-браузеры и серверы взаимодействуют при предоставлении содержимого пользователю. Рассмотрим запрос статической веб-страницы. Пользователь вводит в браузере унифицированный указатель ресурса (Uniform Resource Locator — URL). Браузер генерирует HTTP-запрос к соответствующему веб-серверу. Веб-сервер сопоставляет этот запрос с конкретным файлом, который возвращается браузеру в HTTP-ответе. Заголовок HTTP в ответе указывает тип содержимого. Для этой цели используются многоцелевые расширения почты Интернета (Multipurpose Internet Mail Extensions — MIME). Например, обычный текст ASCII имеет MIME-тип `text/plain`. Исходный код веб-страницы на языке гипертекстовой разметки (Hypertext Markup Language — HTML) имеет MIME-тип `text/html`.

А теперь рассмотрим динамическое содержимое. Предположим, что в Интернет-магазине с применением базы данных сохраняется информация о коммерческой деятельности. Такая информация включает продаваемые товары, цены, наличие, заказы и т.д. Компания желает сделать эту информацию доступной для клиентов через веб-страницы, содержимое которых должно генерироваться динамически, чтобы отражать самую последнюю информацию из базы данных.

На заре развития Интернета сервер мог динамически конструировать страницы, создавая отдельный процесс для обработки каждого клиентского запроса. Для получения необходимой информации процесс открывал подключение к одной или нескольким базам данных. Он связывался с веб-сервером

через общий шлюзовой интерфейс (Common Gateway Interface — CGI), который позволял отдельному процессу читать данные из HTTP-запроса и записывать данные в HTTP-ответ. Для построения программ CGI использовались многочисленные языки, в том числе C, C++ и Perl.

Однако CGI были присущи серьезные проблемы с производительностью. Создание отдельного процесса для каждого клиентского запроса было сопряжено с высокими затратами ресурсов процессора и памяти. Также было неэффективно открывать и закрывать подключение к базе данных для каждого клиентского запроса. Кроме того, программы CGI не были независимыми от платформы. По указанным причинам появились другие методы, в число которых входят сервлеты.

Сервлеты в сравнении с CGI обладают рядом преимуществ. Во-первых, их производительность значительно выше. Сервлеты выполняются в адресном пространстве веб-сервера. Нет необходимости создавать отдельный процесс для обработки каждого запроса клиента. Во-вторых, сервлеты не зависят от платформы, т.к. они написаны на Java. В-третьих, для защиты ресурсов на сервере можно вводить набор ограничений. Наконец, в-четвертых, сервлетам доступна полная функциональность библиотек классов Java. Они могут взаимодействовать с другим программным обеспечением через сокеты и механизмы RMI, которые демонстрировались ранее.

Жизненный цикл сервлета

Центральное место в жизненном цикле сервлета занимают три метода: `init()`, `service()` и `destroy()`. Они реализуются каждым сервлетом и вызываются сервером в определенное время. Рассмотрим типичный пользовательский сценарий, чтобы понять, когда вызываются упомянутые методы.

- Предположим, что пользователь вводит унифицированный указатель ресурса (URL) в веб-браузере. Браузер генерирует HTTP-запрос для введенного URL. Затем этот запрос отправляется надлежащему серверу.
- HTTP-запрос принимается веб-сервером, который сопоставляет его с конкретным сервлетом. Сервлет динамически извлекается и загружается в адресное пространство сервера.
- Сервер вызывает метод `init()` сервлета. Этот метод вызывается только при первой загрузке сервлета в память. Сервлету можно передавать параметры инициализации, чтобы он мог себя конфигурировать.
- Сервер вызывает метод `service()` сервлета для обработки HTTP-запроса. Сервлет способен читать данные, предоставленные в HTTP-запросе. Он также может сформировать HTTP-ответ для клиента. Сервлет остается в адресном пространстве сервера и доступен для обработки любых других HTTP-запросов, полученных от клиентов. Метод `service()` вызывается для каждого HTTP-запроса.

- Наконец, сервер может принять решение выгрузить сервлет из своей памяти. Алгоритмы, по которым принимается такое решение, специфичны для каждого сервера. Сервер вызывает метод `destroy()`, чтобы освободить любые ресурсы вроде файловых дескрипторов, выделенные для сервлета. Важные данные могут быть сохранены в постоянном хранилище. Память, выделенная для сервлета и его объектов, затем может быть подвергнута сборке мусора.

Варианты разработки сервлетов

Для экспериментирования с сервлетами потребуется доступ к контейнеру/серверу сервлетов. Двумя популярными вариантами являются Glassfish и Apache Tomcat. В настоящей главе применяется Apache Tomcat — продукт с открытым исходным кодом, поддерживаемый Apache Software Foundation.

Хотя такие IDE-среды, как NetBeans и Eclipse, очень полезны и могут упростить создание сервлетов, в главе они не используются. Способы разработки и развертывания сервлетов отличаются в разных IDE-средах, и в книге попросту невозможно рассмотреть каждую такую среду. Кроме того, многие читатели будут применять инструменты командной строки, а не какую-то IDE-среду. Таким образом, если вы используете IDE-среду, то для выяснения, каким образом разрабатывать и развертывать сервлеты, должны обратиться в сопровождающую ее документацию. По указанной причине приводимые в главе инструкции предполагают применение только инструментов командной строки. В результате они будут пригодны практически для всех читателей.

Как уже упоминалось, в примерах главы используется контейнер Tomcat, который предлагает простой, но эффективный способ экспериментирования с сервлетами посредством инструментов командной строки. Он доступен в различных средах программирования. Кроме того, из-за применения только инструментов командной строки вам не придется загружать и устанавливать IDE-среду лишь в целях экспериментирования с сервлетами. Тем не менее, важно понимать, что даже в случае разработки в среде с другим контейнером сервлетов представленные здесь концепции по-прежнему актуальны. Просто механизм подготовки сервлета к тестированию будет немного отличаться.

Помните! Инструкции по разработке и развертыванию сервлетов, приводимые в главе, основаны на Tomcat и задействуют только инструменты командной строки. Если вы используете IDE-среду и/или другой контейнер/сервер сервлетов, тогда обратитесь к документации по имеющейся среде.

Использование Tomcat

Контейнер Tomcat содержит библиотеки классов, документацию и поддержку времени выполнения, которые понадобятся для создания и тестирования сервлетов. Доступно несколько версий Tomcat, и на момент написания книги последней выпущенной версией была 10.0.7, на которую и будут ориентироваться последующие инструкции. Версия Tomcat 10.0.7 применяется

здесь из-за того, что она является современной версией Tomcat, пригодной для очень широкого круга читателей. Загрузить Tomcat можно из веб-сайта `tomcat.apache.org`. Потребуется выбрать версию, которая подходит для существующей среды.

В примерах предполагается работа 64-разрядной среде Windows. Если 64-разрядная версия Tomcat 10.0.7 была распакована непосредственно из корневого каталога, то расположение по умолчанию будет таким:

```
C:\apache-tomcat-10.0.7
```

Именно здесь предполагается нахождение Tomcat 10.0.7 в рассматриваемых далее примерах. Если контейнер Tomcat загружен в другое место (либо используется другая версия Tomcat), тогда придется внести соответствующие изменения в примеры. Может потребоваться указать в переменной среды `JAVA_HOME` каталог верхнего уровня, где установлен комплект Java Development Kit.

На заметку! Для всех каталогов в этом разделе предполагается применение версии Tomcat 10.0.7. В случае установки другой версии Tomcat понадобится скорректировать имена каталогов и пути, чтобы они соответствовали используемым в установленной версии.

После установки Tomcat запускается с помощью файла `startup.bat` в подкаталоге `bin` внутри каталога `apache-tomcat-10.0.7`. Останавливается Tomcat посредством файла `shutdown.bat` в подкаталоге `bin`.

Классы и интерфейсы, необходимые для создания сервлетов, содержатся в архиве `servlet-api.jar`, который находится в показанном ниже каталоге:

```
C:\apache-tomcat-10.0.7\lib
```

Чтобы сделать `servlet-api.jar` доступным, нужно обновить переменную среды `CLASSPATH` для включения в нее следующего пути к файлу:

```
C:\apache-tomcat-10.0.7\lib\servlet-api.jar
```

В качестве альтернативы данный файл можно указать при компиляции сервлетов. Скажем, приведенная далее команда компилирует первый пример сервлета:

```
javac HelloServlet.java -classpath "C:\apache-tomcat-10.0.7\lib\servlet-api.jar"
```

После компиляции сервлета необходимо позволить Tomcat найти его. Для этого сервлет понадобится поместить в подкаталог внутри каталога `webapps` контейнера Tomcat и ввести его имя в файле `web.xml`. Ради простоты в примерах применяется каталог и файл `web.xml`, которые Tomcat предоставляет для собственных примеров сервлетов. В итоге не придется создавать файлы или каталоги только для того, чтобы поэкспериментировать с примерами сервлетов. Ниже описана процедура, которой нужно будет следовать.

Первым делом необходимо скопировать файл класса сервлета в следующий каталог:

```
C:\apache-tomcat-10.0.7\webapps\examples\WEB-INF\classes
```

Затем понадобится добавить имя и сопоставление сервлета в файл web.xml, находящийся в показанном ниже каталоге:

```
C:\apache-tomcat-10.0.7\webapps\examples\WEB-INF
```

Скажем, для первого примера под названием HelloServlet в раздел определения сервлетов будут добавлены следующие строки:

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>HelloServlet</servlet-class>
</servlet>
```

А в раздел сопоставления сервлетов — такие строки:

```
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/servlets/servlet/HelloServlet</url-pattern>
</servlet-mapping>
```

Та же самая общая процедура используется во всех примерах.

Простой сервлет

Чтобы ознакомиться с ключевыми концепциями сервлета, имеет смысл начать с создания и тестирования простого сервлета. Ниже описаны основные шаги.

- Создать и скомпилировать исходный код сервлета. Затем скопировать файл класса сервлета в надлежащий каталог и добавить имя и отображение сервлета в соответствующий файл web.xml.
- Запустить Tomcat.
- Запустить веб-браузер и запросить сервлет.

Давайте рассмотрим все перечисленные шаги более подробно.

Создание и компиляция исходного кода сервлета

Первым делом нужно создать файл по имени HelloServlet.java со следующим кодом:

```
import java.io.*;
import jakarta.servlet.*;

public class HelloServlet extends GenericServlet {

  public void service(ServletRequest request,
                      ServletResponse response)
                      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>Hello!");
    pw.close();
  }
}
```

Проанализируем код. Прежде всего, обратите внимание, что в нем импортируется пакет `jakarta.servlet`, который содержит классы и интерфейсы, необходимые для создания сервлетов. Позже в главе вы узнаете о нем больше. Затем определяется класс `HelloServlet` как подкласс `GenericServlet`. Класс `GenericServlet` обеспечивает функциональность, упрощающую создание сервлета. Например, он предоставляет версии методов `init()` и `destroy()`, которые можно применять в том виде как есть. Понадобится реализовать только метод `service()`.

Внутри класса `HelloServlet` переопределяется метод `service()`, унаследованный из `GenericServlet`, который обрабатывает запросы от клиента. Обратите внимание, что первый аргумент представляет собой объект `ServletRequest`, который дает сервлету возможность читать данные, предоставленные через клиентский запрос. Второй аргумент — объект `ServletResponse`, позволяющий сервлету формировать ответ для клиента.

Вызов `setContentType()` устанавливает MIME-тип HTTP-ответа. Здесь MIME-типом является `text/html`, указывающий на то, что браузер должен интерпретировать содержимое как исходный код HTML.

Далее метод `getWriter()` получает поток `PrintWriter`. Все, что записывается в данный поток, отправляется клиенту в виде части HTTP-ответа. Затем в качестве HTTP-ответа с помощью `println()` записывается простой исходный код HTML. Готовый исходный код необходимо скомпилировать и поместить файл `HelloServlet.class` в соответствующий каталог Tomcat, как было описано в предыдущем разделе. Кроме того, `HelloServlet` понадобится добавить в файл `web.xml`, как объяснялось выше.

Запуск Tomcat

Запустите Tomcat согласно приведенным ранее инструкциям. Контейнер Tomcat должен функционировать, прежде чем будет предпринята попытка выполнить сервлет.

Запуск веб-браузера и запрашивание сервлета

Запуск веб-браузера и введите показанный ниже URL:

```
http://localhost:8080/examples/servlets/servlet/HelloServlet
```

В качестве альтернативы можно ввести следующий URL:

```
http://127.0.0.1:8080/examples/servlets/servlet/HelloServlet
```

Так поступать разрешено, поскольку 127.0.0.1 определено как IP-адрес локальной машины.

Вы увидите вывод сервлета в области отображения браузера. Он будет содержать строку `Hello!` с полужирным начертанием.

Servlet API

Классы и интерфейсы, которые необходимы для создания сервлетов, описанных в этой главе, содержатся в двух пакетах: `jakarta.servlet` и `jakarta.`

servlet.http. Они образуют ядро Servlet API. Имейте в виду, что упомянутые пакеты не входят в состав основных пакетов Java и потому они не включены в Java SE. Взамен они предоставляются реализацией сервлета, в данном случае Tomcat.

Servlet API постоянно развивается и совершенствуется. Спецификация сервлетов, поддерживаемая Tomcat 10.0.7, имеет версию 5.0. На момент написания книги это была самая последняя спецификация сервлетов, так что она используется в настоящем издании. Однако поскольку в главе обсуждается ядро Servlet API, представленная здесь информация применима к большинству версий спецификации сервлетов (и Tomcat), за исключением следующего момента.

Прежде чем продолжить, необходимо сделать важное замечание. До выхода спецификации сервлетов версии 5 пакеты Servlet API начинались с javax, а не с jakarta. Таким образом, в случае применения версии Tomcat, предшествующей 10 (или реализации сервлетов на основе спецификации до версии 5), потребуется изменить все ссылки в примерах программ с jakarta на javax. Скажем, jakarta.servlet превратится в javax.servlet.

Помните! Для реализаций сервлетов, которые основаны на версиях спецификаций, предшествующих 5, Servlet API будет находиться в пакетах javax, а не jakarta.

Пакет jakarta.servlet

Пакет jakarta.servlet содержит несколько интерфейсов и классов, образующих инфраструктуру, в которой функционируют сервлеты. В табл. 36.1 кратко описаны ключевые интерфейсы, предоставляемые пакетом jakarta.servlet. Наиболее важным из них является Servlet. Все сервлеты должны реализовывать интерфейс Servlet или расширять реализующий его класс. Интерфейсы ServletRequest и ServletResponse тоже очень важны.

Таблица 36.1. Основные интерфейсы в пакете jakarta.servlet

Интерфейс	Описание
Servlet	Объявляет методы жизненного цикла сервлета
ServletConfig	Позволяет сервлетам получать параметры инициализации
ServletContext	Позволяет сервлетам регистрировать события в журнале и получать доступ к информации, касающейся их среды
ServletRequest	Используется для чтения данных из клиентского запроса
ServletResponse	Используется для записи данных в клиентский ответ

В табл. 36.2 кратко описаны основные классы, предоставляемые пакетом jakarta.servlet.

Таблица 36.2. Основные классы в пакете `jakarta.servlet`

Класс	Описание
`GenericServlet`	Реализует интерфейсы `Servlet` и `ServletConfig`
`ServletInputStream`	Инкапсулирует поток ввода для чтения запросов от клиента
`ServletOutputStream`	Инкапсулирует поток вывода для записи ответов клиенту
`ServletException`	Указывает на то, что в сервлете произошла ошибка
`UnavailableException`	Указывает на то, что сервлет недоступен

Ниже интерфейсы и классы рассматриваются более подробно.

Интерфейс `Servlet`

Все сервлеты должны реализовывать интерфейс `Servlet`, в котором объявлены методы `init()`, `service()` и `destroy()`, вызываемые сервером в течение жизненного цикла сервлета. Также предоставляется метод, который позволяет сервлету получать любые параметры инициализации. Методы, определенные в интерфейсе `Servlet`, показаны в табл. 36.3.

Таблица 36.3. Методы, определенные в интерфейсе `Servlet`

Метод	Описание
`void destroy()`	Вызывается при выгрузке сервлета
`ServletConfig getServletConfig()`	Возвращает объект `ServletConfig`, который содержит параметры инициализации
`String getServletInfo()`	Возвращает строку, описывающую сервлет
`void init (ServletConfig sc) throws ServletException`	Вызывается при инициализации сервлета. Параметры инициализации для сервлета могут быть получены из `sc`. Если инициализировать сервлет невозможно, тогда должно быть сгенерировано исключение `ServletException`
`void service (ServletRequest req, ServletResponse res) throws ServletException, IOException`	Вызывается для обработки запроса от клиента. Запрос от клиента может быть прочитан из `req`. Ответ клиенту может быть записан в `res`. В случае возникновения ошибки сервлета или ввода-вывода генерируется исключение

Методы `init()`, `service()` и `destroy()` являются методами жизненного цикла сервлета. Они вызываются сервером. Метод `getServletConfig()` вы-

зывается сервлетом для получения параметров инициализации. Разработчик сервлета переопределяет метод getServletInfo() с целью предоставления строки с полезной информацией (например, номером версии). Этот метод также вызывается сервером.

Интерфейс ServletConfig

Интерфейс ServletConfig позволяет сервлету получать данные конфигурации во время его загрузки. Методы, объявленные в интерфейсе ServletConfig, кратко описаны в табл. 36.4.

Таблица 36.4. Методы, определенные в интерфейсе ServletConfig

Метод	Описание
ServletContext getServletContext()	Возвращает контекст сервлета
String getInitParameter (String param)	Возвращает значение параметра инициализации по имени param
Enumeration<String> getInitParameterNames()	Возвращает перечисление с именами всех параметров инициализации
String getServletName()	Возвращает имя вызывающего сервлета

Интерфейс ServletContext

Интерфейс ServletContext позволяет сервлетам получать информацию, касающуюся их среды. Ряд методов интерфейса ServletContext кратко описан в табл. 36.5.

Таблица 36.5. Избранные методы, определенные в интерфейсе ServletContext

Метод	Описание
Object getAttribute (String attr)	Возвращает значение атрибута сервера по имени attr
String getMimeType (String file)	Возвращает MIME-тип файла file
String getRealPath (String vpath)	Возвращает реальный (т.е. абсолютный) путь, который соответствует относительному пути vpath
String getServerInfo()	Возвращает информацию о сервере
void log(String s)	Записывает s в журнал сервлета
void log(String s, Throwable e)	Записывает s и трассировку стека для e в журнал сервлета
void setAttribute (String attr, Object val)	Устанавливает для атрибута, указанного в attr, значение, переданное в val

Интерфейс ServletRequest

Интерфейс ServletRequest позволяет сервлету получать информацию о клиентском запросе. Ряд методов интерфейса ServletRequest кратко описан в табл. 36.6.

Таблица 36.6. Избранные методы, определенные в интерфейсе ServletRequest

Метод	Описание
`Object getAttribute (String attr)`	Возвращает значение атрибута по имени attr
`String getCharacterEncoding()`	Возвращает кодировку символов в запросе
`int getContentLength()`	Возвращает размер запроса. Если размер не может быть определен, тогда возвращается значение –1
`String getContentType()`	Возвращает тип запроса. Если тип не может быть определен, тогда возвращается значение null
`ServletInputStream getInputStream() throws IOException`	Возвращает объект ServletInputStream, который можно использовать для чтения двоичных данных из запроса. Если ранее на этом объекте не вызывался метод getReader(), тогда генерируется исключение IllegalStateException
`String getParameter (String pname)`	Возвращает значение параметра по имени pname
`Enumeration<String> getParameterNames()`	Возвращает перечисление с именами всех параметров для данного запроса
`String[] getParameterValues (String name)`	Возвращает массив, содержащий значения, которые ассоциированы с параметром, указанным в name
`String getProtocol()`	Возвращает описание протокола
`BufferedReader getReader() throws IOException`	Возвращает буферизированное средство чтения, которое можно использовать для чтения текста из запроса. Если ранее на этом объекте не вызывался метод getInputStream(), тогда генерируется исключение IllegalStateException
`String getRemoteAddr()`	Возвращает строковый эквивалент для IP-адреса клиента

Окончание табл. 36.6

Метод	Описание
`String getRemoteHost()`	Возвращает строковый эквивалент для имени хоста клиента
`String getScheme()`	Возвращает схему передачи URL, используемую для запроса (например, "http", "ftp")
`String getServerName()`	Возвращает имя сервера
`int getServerPort()`	Возвращает номер порта

Интерфейс ServletResponse

Интерфейс ServletResponse позволяет сервлету формировать ответ для клиента. Некоторые методы интерфейса ServletResponse кратко описаны в табл. 36.7.

Таблица 36.7. Избранные методы, определенные в интерфейсе ServletResponse

Метод	Описание
`String getCharacterEncoding()`	Возвращает кодировку символов для ответа
`ServletOutputStream getOutputStream() throws IOException`	Возвращает объект ServletOutputStream, который можно использовать для записи двоичных данных в ответ. Если ранее на этом объекте не вызывался метод getWriter(), тогда генерируется исключение IllegalStateException
`PrintWriter getWriter() throws IOException`	Возвращает объект PrintWriter, который можно использовать для записи символьных данных в ответ. Если ранее на этом объекте не вызывался метод getOutputStream(), тогда генерируется исключение IllegalStateException
`void setContentLength (int size)`	Устанавливает длину содержимого для ответа в size
`void setContentType (String type)`	Устанавливает тип содержимого для ответа в type

Класс GenericServlet

Класс GenericServlet предоставляет реализации базовых методов жизненного цикла сервлета. Он реализует интерфейсы Servlet и ServletConfig. Кроме того, доступен метод для добавления строки в журнальный файл сервера.

Вот сигнатуры данного метода:

```
void log(String s)
void log(String s, Throwable e)
```

В s указывается строка, добавляемая в журнал, а в e — возникшее исключение.

Класс ServletInputStream

Класс ServletInputStream расширяет InputStream. Он реализован контейнером сервлетов и предлагает поток ввода, который разработчик сервлета может применять для чтения данных из клиентского запроса. В дополнение к методам ввода, унаследованным из InputStream, предоставляется метод для чтения байтов из потока:

```
int readLine(byte[] buffer, int offset, int size) throws IOException
```

В buffer указывается массив, куда помещаются size байтов, начиная со смещения offset. Метод возвращает фактическое количество прочитанных байтов или -1, если встретился конец потока.

Класс ServletOutputStream

Класс ServletOutputStream расширяет OutputStream. Он реализован контейнером сервлетов и предлагает поток вывода, который разработчик сервлета может использовать для записи данных в ответ клиента. Вдобавок к методам вывода, предоставляемым OutputStream, в нем также определены методы print() и println(), которые выводят данные в поток.

Классы исключений сервлетов

В пакете jakarta.servlet определены два исключения. Первое, ServletException, указывает на то, что возникла проблема с сервлетом. Второе, UnavailableException, расширяет ServletException и указывает на то, что сервлет недоступен.

Чтение параметров сервлета

В интерфейсе ServletRequest есть методы, позволяющие читать имена и значения параметров, включенных в клиентский запрос. Будет разработан сервлет, иллюстрирующий их применение. В состав пример входят два файла. Веб-страница определена в PostParameters.html, а сервлет — в PostParametersServlet.java.

Ниже показано содержимое файла PostParameters.html. В нем определяется таблица, содержащая две метки и два текстовых поля. Одной из меток является Employee, а другой — Phone. Также имеется кнопка отправки. Обратите внимание, что в параметре action дескриптора <form> указан URL, который идентифицирует сервлет для обработки HTTP-запроса POST.

```
<html>
<body>
<center>
<form name="Form1"
      method="post"
      action="http://localhost:8080/examples/servlets/servlet/
PostParametersServlet">
  <table>
    <tr>
      <td><B>Employee</td>
      <td><input type=textbox name="e" size="25" value=""></td>
    </tr>
    <tr>
      <td><B>Phone</td>
      <td><input type=textbox name="p" size="25" value=""></td>
    </tr>
  </table>
  <input type=submit value="Submit">
</body>
</html>
```

Далее приведен исходный код `PostParametersServlet.java`. Метод `service()` переопределяется для обработки клиентских запросов. Метод `getParameterNames()` возвращает перечисление с именами параметров, которое обрабатывается в цикле. Имена и значения параметров выводятся клиенту. Значение параметра получается с помощью метода `getParameter()`.

```
import java.io.*;
import java.util.*;
import jakarta.servlet.*;

public class PostParametersServlet extends GenericServlet {

  public void service(ServletRequest request, ServletResponse response)
                      throws ServletException, IOException {

    // Получить экземпляр PrintWriter.
    PrintWriter pw = response.getWriter();

    // Получить перечисление с именами параметров.
    Enumeration<String> e = request.getParameterNames();

    // Отобразить имена и значения параметров.
    while(e.hasMoreElements()) {
      String pname = e.nextElement();
      pw.print(pname + " = ");
      String pvalue = request.getParameter(pname);
      pw.println(pvalue);
    }
    pw.close();
  }
}
```

Скомпилируйте сервлет, после чего скопируйте его в соответствующий каталог и обновите файл `web.xml`, как было описано ранее. Затем выполните перечисленные ниже действия, чтобы протестировать пример сервлета.

- Запустить Tomcat (если он еще не функционирует).
- Отобразить веб-страницу в браузере.
- Ввести в текстовых полях имя и телефонный номер сотрудника.
- Отправить форму на веб-странице.

После выполнения описанных действий браузер отобразит ответ, динамически сгенерированный сервлетом.

Пакет `jakarta.servlet.http`

В предшествующих примерах для иллюстрации базовой функциональности сервлетов использовались классы и интерфейсы, определенные в пакете `jakarta.servlet`, такие как `ServletRequest`, `ServletResponse` и `GenericServlet`. Тем не менее, при работе с HTTP обычно применяются интерфейсы и классы из пакета `jakarta.servlet.http`. Вы увидите, что его функциональность позволяет легко создавать сервлеты, работающие с запросами и ответами HTTP.

В табл. 36.8 кратко описаны интерфейсы, используемые в главе.

Таблица 36.8. Интерфейсы, определенные в пакете `jakarta.servlet.http`

Интерфейс	Описание
`HttpServletRequest`	Позволяет сервлетам читать данные из HTTP-запроса
`HttpServletResponse`	Позволяет сервлетам записывать данные в HTTP-ответ
`HttpSession`	Позволяет читать и записывать данные сеанса

В табл. 36.9 кратко описаны классы, применяемые в главе. Наиболее важным мз них является `HttpServlet`, который разработчики сервлетов обычно расширяют для обработки HTTP-запросов.

Таблица 36.9. Классы, определенные в пакете `jakarta.servlet.http`

Класс	Описание
`Cookie`	Позволяет сохранять информацию о состоянии на клиентской машине
`HttpServlet`	Предоставляет методы для обработки запросов и ответов HTTP

Интерфейс `HttpServletRequest`

Интерфейс `HttpServletRequest` позволяет сервлету получать информацию о клиентском запросе. Некоторые его методы кратко описаны в табл. 36.10.

Таблица 36.10. Избранные методы, определенные в интерфейсе `HttpServletRequest`

Метод	Описание
`String getAuthType()`	Возвращает схему аутентификации
`Cookie[] getCookies()`	Возвращает массив cookie-наборов в запросе
`long getDateHeader (String field)`	Возвращает значение поля заголовка даты по имени `field`
`String getHeader (String field)`	Возвращает значение поля заголовка по имени `field`
`Enumeration<String> getHeaderNames()`	Возвращает перечисление с именами заголовков
`int getIntHeader (String field)`	Возвращает эквивалент типа `int` поля заголовка по имени `field`
`String getMethod()`	Возвращает HTTP-метод для запроса
`String getPathInfo()`	Возвращает информацию о пути, который находится после пути к сервлету и перед строкой запроса в URL
`String getPathTranslated()`	Возвращает информацию о пути, который находится после пути к сервлету и перед строкой запроса в URL, после его преобразования в реальный путь
`String getQueryString()`	Возвращает строку запроса в URL
`String getRemoteUser()`	Возвращает имя пользователя, выдавшего запрос
`String getRequestedSessionId()`	Возвращает идентификатор сеанса
`String getRequestURI()`	Возвращает URI
`StringBuffer getRequestURL()`	Возвращает URL
`String getServletPath()`	Возвращает часть URL, которая идентифицирует сервлет
`HttpSession getSession()`	Возвращает сеанс для запроса. Если сеанс не существует, тогда он создается и затем возвращается
`HttpSession getSession (boolean new)`	Если `new` равно `true` и сеанс не существует, тогда создает и возвращает сеанс для запроса, а иначе возвращает для него существующий сеанс

Окончание табл. 36.10

Метод	Описание
`boolean isRequestedSessionIdFromCookie()`	Возвращает `true`, если cookie-набор содержит идентификатор сеанса, или `false` в противном случае
`boolean isRequestedSessionIdFromURL()`	Возвращает `true`, если URL содержит идентификатор сеанса, или `false` в противном случае
`boolean isRequestedSessionIdValid()`	Возвращает `true`, если запрошенный идентификатор сеанса является допустимым в контексте текущего сеанса

Интерфейс `HttpServletResponse`

Интерфейс `HttpServletResponse` позволяет сервлету формировать HTTP-ответ для клиента. В нем определено несколько констант, соответствующих различным кодам состояния, которые могут быть назначены HTTP-ответу. Например, `SC_OK` указывает, что HTTP-запрос выполнен успешно, а `SC_NOT_FOUND` — что запрошенный ресурс недоступен. В табл. 36.11 кратко описано несколько методов интерфейса `HttpServletResponse`.

Таблица 36.11. Избранные методы, определенные в интерфейсе `HttpServletResponse`

Метод	Описание
`void addCookie(Cookie cookie)`	Добавляет cookie-набор в HTTP-ответ
`boolean containsHeader (String field)`	Возвращает `true`, если заголовок HTTP-ответа содержит поле по имени `field`
`String encodeURL(String url)`	Определяет, должен ли идентификатор сеанса быть закодирован в URL, указанном с помощью `url`. Если это так, тогда возвращается измененная версия `url`. В противном случае возвращается `url` без изменений. Данным методом должны обрабатываться все URL, сгенерированные сервлетом
`String encodeRedirectURL(String url)`	Определяет, должен ли идентификатор сеанса быть закодирован в URL, указанном посредством `url`. Если это так, тогда возвращается измененная версия `url`. В противном случае возвращается `url` без изменений. Данным методом должны обрабатываться все URL, передаваемые `sendRedirect()`

Окончание табл. 36.11

Метод	Описание
`void sendError(int c) throws IOException`	Посылает клиенту код ошибки c
`void sendError(int c, String s) throws IOException`	Посылает клиенту код ошибки c и сообщение s
`void sendRedirect(String url) throws IOException`	Переадресует клиента на `url`
`void setDateHeader(String field, long msec)`	Добавляет в заголовок поле `field` со значением даты, указанным в `msec` (количество миллисекунд, прошедших с полуночи 1 января 1970 года, GMT)
`void setHeader(String field, String value)`	Добавляет в заголовок поле `field` со значением, указанным в `value`
`void setIntHeader(String field, int value)`	Добавляет в заголовок поле `field` со значением, указанным в `value`
`void setStatus(int code)`	Устанавливает код состояния для этого ответа в `code`

Интерфейс `HttpSession`

Интерфейс `HttpSession` позволяет сервлету читать и записывать информацию о состоянии сеанса HTTP. Некоторые из его методов кратко описаны в табл. 36.12. Все они генерируют исключение `IllegalStateException`, если сеанс уже недействителен.

Таблица 36.12. Избранные методы, определенные в интерфейсе `HttpSession`

Метод	Описание
`Object getAttribute (String attr)`	Возвращает значение, которое ассоциировано с именем, переданным в `attr`, или `null`, если имя `attr` не найдено
`Enumeration<String> getAttributeNames()`	Возвращает перечисление имен атрибутов, ассоциированных с сеансом
`long getCreationTime()`	Возвращает время создания (количество миллисекунд, прошедших с полуночи 1 января 1970 года, GMT) вызывающего сеанса
`String getId()`	Возвращает идентификатор сеанса
`long getLastAccessedTime()`	Возвращает время (количество миллисекунд, прошедших с полуночи 1 января 1970 года, GMT), когда клиент последний раз выполнял запрос в вызывающем сеансе

Окончание табл. 36.12

Метод	Описание
`void invalidate()`	Делает недействительным сеанс и удаляет его из контекста
`boolean isNew()`	Возвращает `true`, если сервер создал сеанс, к которому клиент пока не осуществлял доступ
`void removeAttribute (String attr)`	Удаляет из сеанса атрибут, указанный в `attr`
`void setAttribute (String attr, Object val)`	Ассоциирует значение, переданное в `val`, с именем атрибута, указанным в `attr`

Класс Cookie

Класс `Cookie` инкапсулирует *cookie-набор*, который хранится на стороне клиента и содержит информацию о состоянии. Cookie-наборы полезны для отслеживания действий пользователей. Например, предположим, что пользователь посещает Интернет-магазин. Если cookie-набор сохранит имя пользователя, адрес и другую информацию, то пользователю не придется вводить такие данные при каждом посещении магазина.

Сервлет может записать cookie-набор на машину пользователя с помощью метода `addCookie()` интерфейса `HttpServletResponse`. Затем данные для cookie-набора включаются в заголовок HTTP-ответа, отправляемого браузеру.

Имена и значения cookie-наборов хранятся на машине пользователя. Для каждого cookie-набора может быть сохранена определенная информация, в том числе:

- имя cookie-набора;
- значение cookie-набора;
- срок действия cookie-набора;
- домен и путь cookie-набора.

Срок действия определяет, когда cookie-набор будет удален из машины пользователя. Если срок действия cookie-набора не назначен явно, тогда он удаляется по завершении текущего сеанса браузера.

Домен и путь cookie-набора определяют, когда он включается в заголовок HTTP-запроса. Если пользователь вводит URL, домен и путь которого соответствуют таким значениям, то cookie-набор передается веб-серверу. В противном случае он не передается.

В классе `Cookie` имеется один конструктор со следующей сигнатурой:

```
Cookie(String name, String value)
```

В качестве аргументов конструктору передаются имя и значение cookie-набора. Методы класса `Cookie` кратко описаны в табл. 36.13.

Таблица 36.13. Методы, определенные в классе Cookie

Метод	Описание
`Object clone()`	Возвращает копию этого объекта
`String getComment()`	Возвращает комментарий
`String getDomain()`	Возвращает домен
`int getMaxAge()`	Возвращает максимальный срок действия (в секундах)
`String getName()`	Возвращает имя
`String getPath()`	Возвращает путь
`boolean getSecure()`	Возвращает `true`, если cookie-набор безопасный, или `false` в противном случае
`String getValue()`	Возвращает значение
`int getVersion()`	Возвращает версию
`boolean isHttpOnly()`	Возвращает `true`, если cookie-набор имеет атрибут `HttpOnly`
`void setComment(String c)`	Устанавливает комментарий в `c`
`void setDomain(String d)`	Устанавливает домен в `d`
`void setHttpOnly (boolean httpOnly)`	Если `httpOnly` равно `true`, тогда в cookie-набор добавляется атрибут `HttpOnly`, а если `httpOnly` равно `false`, то атрибут `HttpOnly` удаляется
`void setMaxAge(int secs)`	Устанавливает максимальный срок действия cookie-набора в `secs` — количество секунд, после истечения которых cookie-набор удаляется
`void setPath(String p)`	Устанавливает путь в `p`
`void setSecure (boolean secure)`	Устанавливает флаг безопасности в `secure`
`void setValue(String v)`	Устанавливает значение в `v`
`void setVersion(int v)`	Устанавливает версию в `v`

Класс `HttpServlet`

Класс `HttpServlet` расширяет `GenericServlet`. Он обычно используется при разработке сервлетов, которые получают и обрабатывают HTTP-запросы. Методы, определенные в классе `HttpServlet`, кратко описаны в табл. 36.14.

Таблица 36.14. Методы, определенные в классе `HttpServlet`

Метод	Описание
`void doDelete(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Обрабатывает HTTP-запрос `DELETE`
`void doGet(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Обрабатывает HTTP-запрос `GET`
`void doHead(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Обрабатывает HTTP-запрос `HEAD`
`void doOptions(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Обрабатывает HTTP-запрос `OPTIONS`
`void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Обрабатывает HTTP-запрос `POST`
`void doPut(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Обрабатывает HTTP-запрос `PUT`
`void doTrace(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Обрабатывает HTTP-запрос `TRACE`
`long getLastModified (HttpServletRequest req)`	Возвращает время (количество миллисекунд, прошедших с полуночи 1 января 1970 года, GMT), когда запрошенный ресурс модифицировался в последний раз
`void service(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Вызывается сервером, когда поступает HTTP-запрос для этого сервлета. Аргументы предоставляют доступ к запросу и ответу HTTP соответственно

Обработка запросов и ответов HTTP

Класс `HttpServlet` предлагает специализированные методы для обработки HTTP-запросов различных видов. Разработчик сервлета обычно переопределяет один из таких методов: `doDelete()`, `doGet()`, `doHead()`, `doOptions()`, `doPost()`, `doPut()` и `doTrace()`. Полное описание всех видов HTTP-запросов

выходит за рамки книги. Однако запросы GET и POST обычно применяются при обработке данных, вводимых в форму, и потому в настоящем разделе представлены примеры таких случаев.

Обработка HTTP-запросов GET

Здесь будет разработан сервлет, который обрабатывает HTTP-запрос GET. Сервлет вызывается при отправке формы на веб-странице. В состав примера входят два файла. Веб-страница определяется в ColorGet.html, а сервлет — в ColorGetServlet.java. Ниже показан исходный код HTML для ColorGet.html. В нем определяется форма с элементом выбора и кнопкой отправки. Обратите внимание, что в параметре action дескриптора <form> указан URL, который идентифицирует сервлет для обработки HTTP-запроса GET.

```
<html>
<body>
  <center>
    <form name="Form1" action="http://localhost:8080/examples/servlets/
servlet/ColorGetServlet">
    <B>Color:</B>
    <select name="color" size="1">
      <option value="Red">Red</option>
      <option value="Green">Green</option>
      <option value="Blue">Blue</option>
    </select>
    <br><br>
    <input type=submit value="Submit">
  </form>
</body>
</html>
```

Далее приведен исходный код для ColorGetServlet.java. Метод doGet() переопределяется для обработки любых HTTP-запросов GET, отправляемых этому сервлету. В нем с помощью метода getParameter() интерфейса HttpServletRequest получается выбор, сделанный пользователем, после чего формируется ответ.

```
import java.io.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class ColorGetServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                    throws ServletException, IOException {
    String color = request.getParameter("color");
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>The selected color is: ");  // Выбранный цвет
    pw.println(color);
    pw.close();
  }
}
```

Скомпилируйте сервлет. Скопируйте его в соответствующий каталог и обновите файл web.xml, как объяснялось ранее. Затем выполните следующие действия, чтобы протестировать пример сервлета.

- Запустить Tomcat, если он еще не функционирует.
- Отобразить веб-страницу в браузере.
- Выбрать цвет.
- Отправить форму на веб-странице.

После выполнения перечисленных выше шагов браузер отобразит ответ, динамически сгенерированный сервлетом.

И еще один момент: параметры HTTP-запроса GET включаются как часть URL, отправляемого веб-серверу. Предполагая, что пользователь выбрал красный цвет и отправил форму, вот как будет выглядеть URL, отправленный из браузера серверу:

```
http://localhost:8080/examples/servlets/servlet/ColorGetServlet?color=Red
```

Символы справа от вопросительного знака образуют *строку запроса.*

Обработка HTTP-запросов POST

Здесь будет создан сервлет, который обрабатывает HTTP-запрос POST. Сервлет вызывается при отправке формы на веб-странице. В состав примера входят два файла. Веб-страница определяется в ColorPost.html, а сервлет — в ColorPostServlet.java.

Ниже показан исходный код HTML для ColorPost.html. Он идентичен ColorGet.html за исключением того, что в параметре method дескриптора <form> явно указано, что должен использоваться метод POST, а в параметре action дескриптора <form> задан другой сервлет.

```
<html>
<body>
  <center>
  <form name="Form1" method="post"
       action="http://localhost:8080/examples/servlets/servlet/
ColorPostServlet">
    <B>Color:</B>
    <select name="color" size="1">
      <option value="Red">Red</option>
      <option value="Green">Green</option>
      <option value="Blue">Blue</option>
    </select>
    <br><br>
    <input type=submit value="Submit">
  </form>
</body>
</html>
```

Далее представлен исходный код для ColorPostServlet.java. Метод doPost() переопределен с целью обработки любых HTTP-запросов POST, от-

правляемых этому сервлету. В нем применяется метод `getParameter()` интерфейса `HttpServletRequest` для получения выбора, сделанного пользователем. Затем формируется ответ.

```
import java.io.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class ColorPostServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
                     throws ServletException, IOException {
    String color = request.getParameter("color");
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>The selected color is: ");  // Выбранный цвет
    pw.println(color);
    pw.close();
  }
}
```

Скомпилируйте сервлет и выполните шаги, описанные в предыдущем разделе, чтобы его протестировать.

На заметку! Параметры HTTP-запроса `POST` не включаются в URL, отправляемый веб-серверу. В рассмотренном примере URL, отправленный из браузера серверу, имеет вид `http://localhost:8080/examples/servlets/servlet/ColorPostServlet`. Имена и значения параметров передаются в теле HTTP-запроса.

Использование cookie-наборов

Теперь разработаем сервлет, иллюстрирующий использование cookie-наборов. Сервлет вызывается при отправке формы на веб-странице. В состав примера входят три файла, перечисленные в табл. 36.15.

Таблица 36.15. Три файла из примера сервлета

Файл	Описание
`AddCookie.html`	Позволяет пользователю указывать значение для cookie-набора по имени `MyCookie`
`AddCookieServlet.java`	Обрабатывает отправку формы на веб-странице `AddCookie.html`
`GetCookiesServlet.java`	Отображает значения cookie-набора

Ниже показан исходный код HTML для `AddCookie.html`. Страница содержит текстовое поле, в котором можно вводить значение. На странице также предусмотрена кнопка отправки, щелчок на которой приводит к отправке значения в текстовом поле сервлету `AddCookieServlet` через HTTP-запрос `POST`.

```
<html>
<body>
  <center>
  <form name="Form1" method="post"
       action="http://localhost:8080/examples/servlets/servlet/
AddCookieServlet">
    <B>Enter a value for MyCookie:</B>
    <input type=textbox name="data" size=25 value="">
    <input type=submit value="Submit">
  </form>
</body>
</html>
```

Далее представлен исходный код для `AddCookieServlet.java`. Он получает значение параметра по имени `"data"`. Затем он создает объект `Cookie`, который имеет имя `"MyCookie"` и содержит значение параметра `"data"`. Этот cookie-набор добавляется в заголовок HTTP-ответа с помощью метода `addCookie()` и в браузер записывается сообщение обратной связи.

```
import java.io.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;
public class AddCookieServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
                     throws ServletException, IOException {
    // Получить параметр из HTTP-запроса.
    String data = request.getParameter("data");
    // Создать cookie-набор.
    Cookie cookie = new Cookie("MyCookie", data);
    // Добавить cookie-набор в HTTP-ответ.
    response.addCookie(cookie);
    // Записать вывод в браузер.
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>MyCookie has been set to");  // MyCookie установлен
    pw.println(data);
    pw.close();
  }
}
```

Ниже приведен исходный код для `GetCookiesServlet.java`. В нем вызывается метод `getCookies()` для чтения всех cookie-наборов, включенных в HTTP-запрос `GET`. Имена и значения cookie-наборов затем записываются в HTTP-ответ. Обратите внимание, что для получения этой информации вызываются методы `getName()` и `getValue()`.

```
import java.io.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;
public class GetCookiesServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                    throws ServletException, IOException {
```

```
    // Получить cookie-наборы из заголовка HTTP-запроса.
    Cookie[] cookies = request.getCookies();

    // Отобразить полученные cookie-наборы.
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>");
    for(int i = 0; i < cookies.length; i++) {
      String name = cookies[i].getName();
      String value = cookies[i].getValue();
      pw.println("name = " + name +
        "; value = " + value);
    }
    pw.close();
  }
}
```

Скомпилируйте сервлет. Скопируйте его в соответствующий каталог и обновите файл `web.xml`, как объяснялось ранее. Затем выполните следующие действия, чтобы протестировать пример сервлета.

- Запустить Tomcat, если он еще не функционирует.
- Отобразить веб-страницу `AddCookie.html` в браузере.
- Ввести значение для `MyCookie`.
- Отправить форму на веб-странице.

После выполнения этих шагов браузер отобразит сообщение обратной связи.

Запросите в браузере такой URL:

```
http://localhost:8080/examples/servlets/servlet/GetCookiesServlet
```

В браузере отобразятся имя и значение cookie-набора.

В рассмотренном примере срок действия cookie-набора не устанавливается явно с помощью метода `setMaxAge()` класса `Cookie`. Таким образом, срок действия cookie-набора истекает при завершении сеанса браузера. Можете поэкспериментировать с методом `setMaxAge()` и удостовериться в том, что cookie-набор сохраняется на машине клиента.

Отслеживание сеансов

HTTP — протокол, не запоминающий состояние. Каждый запрос не зависит от предыдущего. Тем не менее, в некоторых приложениях необходимо сохранять информацию о состоянии, чтобы можно было собирать информацию из нескольких взаимодействий между браузером и сервером. Сеансы предоставляют такой механизм.

Сеанс может быть создан с применением метода `getSession()` интерфейса `HttpServletRequest`, который возвращает объект `HttpSession`. Этот объект способен хранить набор привязок, ассоциирующих имена с объектами. Управлять такими привязками позволяют методы `setAttribute()`,

getAttribute(), getAttributeNames() и removeAttribute() интерфейса HttpSession. Состояние сеанса совместно используется всеми сервлетами, связанными с клиентом.

В следующем сервлете демонстрируется применение состояния сеанса. Метод getSession() получает текущий сеанс. Если сеанс еще не существует, тогда создается новый сеанс. Метод getAttribute() вызывается для получения объекта, привязанного к имени "date", который представляет собой объект Date, инкапсулирующий дату и время последнего доступа к странице. (Разумеется, при первом доступе к странице такой привязки нет.) Затем создается объект Date, инкапсулирующий текущую дату и время. Метод setAttribute() вызывается для привязки имени "date" к этому объекту.

```
import java.io.*;
import java.util.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class DateServlet extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                    throws ServletException, IOException {

    // Получить объект HttpSession.
    HttpSession hs = request.getSession(true);

    // Получить средство записи.
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.print("<B>");

    // Отобразить дату и время последнего доступа.
    Date date = (Date)hs.getAttribute("date");
    if(date != null) {
      pw.print("Last access: " + date + "<br>");  // Последний доступ
    }

    // Отобразить текущую дату и время.
    date = new Date();
    hs.setAttribute("date", date);
    pw.println("Current date: " + date);          // Текущая дата
  }
}
```

При запрашивании сервлета в первый раз браузер отображает одну строку с информацией о текущей дате и времени. При последующих запросах отображаются две строки: в первой строке находятся дата и время последнего доступа к сервлету, а во второй — текущая дата и время.

ЧАСТЬ

V Приложения

ПРИЛОЖЕНИЕ

A Использование документирующих комментариев Java

Как объяснялось в части I, язык Java поддерживает три вида комментариев. Первые два — // и /* */. Третий вид называется *документирующим комментарием*. Он начинается с последовательности символов /** и заканчивается последовательностью */. Документирующие комментарии позволяют встраивать информацию о программе в саму программу. Затем можно использовать утилиту javadoc (поставляется в составе JDK), чтобы извлечь эту информацию и поместить ее в файл HTML. Документирующие комментарии упрощают документирование программ. Вы наверняка встречались с документацией, в которой применяются такие комментарии, потому что именно так была документирована библиотека Java API. Начиная с версии JDK 9, утилита javadoc включает поддержку модулей.

Дескрипторы javadoc

Утилита javadoc распознает несколько дескрипторов, которые кратко описаны в табл. A.1.

Таблица A.1. Дескрипторы, распознаваемые утилитой javadoc

Дескриптор	Описание
@author	Идентифицирует автора
{@code}	Отображает информацию в том виде как есть, не обрабатывая стили HTML, с использованием шрифта для кода
@deprecated	Указывает, что элемент программы является нерекомендуемым
{@docRoot}	Указывает путь к корневому каталогу текущей документации
@exception	Идентифицирует исключение, генерируемое методом или конструктором

Окончание табл. А.1

Дескриптор	Описание
`@hidden`	Предотвращает отображение элемента в документации
`{@index}`	Указывает термин для индексирования
`{@inheritDoc}`	Наследует комментарий из непосредственного суперкласса
`{@link}`	Вставляет встроенную ссылку на другую тему
`{@linkplain}`	Вставляет встроенную ссылку на другую тему, но ссылка отображается обычным шрифтом
`{@literal}`	Отображает информацию в том виде как есть, не обрабатывая стили HTML
`@param`	Документирует параметр
`@provides`	Документирует службу, предоставляемую модулем
`@return`	Документирует возвращаемое значение метода
`@see`	Указывает ссылку на другую тему
`@serial`	Документирует стандартное сериализируемое поле
`@serialData`	Документирует данные, записываемые методом `writeObject()` или `writeExternal()`
`@serialField`	Документирует компонент `ObjectStreamField`
`@since`	Указывает выпуск, в котором было введено конкретное изменение
`{@summary}`	Документирует сводку по элементу
`{@systemProperty}`	Указывает, что имя является системным свойством
`@throws`	То же, что и `@exception`
`@uses`	Документирует службу, необходимую модулю
`{@value}`	Отображает значение константы, которая должна быть статическим полем
`@version`	Указывает версию элемента программы

Дескрипторы документации, начинающиеся с символа `@`, называются *блочными* (а также *автономными*) и должны использоваться в начале отдельной строки. Дескрипторы, начинающиеся с фигурной скобки, например, `{@code}`, называются *встроенными*, и их можно применять внутри более крупного описания. В документирующем комментарии также допускается использование других стандартных дескрипторов HTML. Однако некоторые дескрипторы,

скажем, заголовки, применяться не должны, т.к. они искажают внешний вид HTML-файла, производимого javadoc.

При подготовке документации по исходному коду документирующие комментарии можно использовать для документирования классов, интерфейсов, полей, конструкторов, методов, пакетов и модулей. Во всех случаях документирующий комментарий должен непосредственно предшествовать документируемому элементу. Некоторые дескрипторы, такие как @see, @since и @deprecated, можно применять для документирования любого элемента. Другие дескрипторы используются только с соответствующими элементами. Далее в приложении будут кратко описаны все дескрипторы.

На заметку! Как и следовало ожидать, возможности javadoc и дескрипторов документирующих комментариев развивались с течением времени, часто в ответ на появление новых функциональных средств Java. Информацию о последних возможностях утилиты javadoc ищите в сопровождающей ее документации.

@author

Дескриптор @author документирует автора элемента программы. Он имеет следующий синтаксис:

```
@author description
```

В description обычно задается имя автора. Чтобы информация в @author была включена в документацию HTML, при запуске javadoc потребуется указать параметр -author.

{@code}

Дескриптор {@code} позволяет вставлять в комментарий текст, такой как фрагмент кода. Затем этот текст отображается в том виде как есть с применением шрифта кода без какой-либо дальнейшей обработки вроде визуализации HTML. Вот его синтаксис, где code-snippet — фрагмент кода:

```
{@code code-snippet}
```

@deprecated

Дескриптор @deprecated указывает, что элемент программы является нерекомендуемым. Имеет смысл с помощью дескрипторов @see или {@link} информировать программиста о доступных альтернативах. Синтаксис выглядит так:

```
@deprecated description
```

В description указывается сообщение с причиной объявления элемента нерекомендуемым. Дескриптор @deprecated можно использовать в документации для полей, методов, конструкторов, классов, модулей и интерфейсов.

{@docRoot}

Дескриптор {@docRoot} задает путь к корневому каталогу текущей документации.

@exception

Дескриптор @exception описывает исключение, генерируемое методом. В настоящее время предпочтительной альтернативой является дескриптор @throws, но @exception по-прежнему поддерживается. Он имеет следующий синтаксис:

```
@exception exception-name explanation
```

В exception-name указывается полное имя исключения, а в explanation — строка с описанием, каким образом может возникнуть исключение. Дескриптор @exception можно применять только в документации для метода или конструктора.

@hidden

Дескриптор @hidden предотвращает появление элемента в документации.

{@index}

Дескриптор {@index} задает элемент, который будет проиндексирован и в результате найден при использовании средства поиска. Вот его синтаксис:

```
{ @index term usage-str }
```

В term указывается элемент (который может быть строкой в кавычках), подлежащий индексации, а в usage-str — необязательное описание применения. Таким образом, показанные ниже дескрипторы @throws {@index} добавляют в индекс термин "error" (ошибка):

```
@throws IOException Возникает при вводе {@index error}.
```

Обратите внимание, что слово "error" по-прежнему отображается как часть описания. Просто теперь оно еще и индексируется. Если включить необязательный параметр usage-str, то описание будет отображаться в индексе и в поле поиска, поясняя, каким образом использовать термин. Например, {@index error Serious execution failure} приводит к появлению описания "Serious execution failure" (Серьезный отказ при выполнении) под термином "error" в индексе и в поле поиска.

{@inheritDoc}

Дескриптор {@inheritDoc} обеспечивает наследование комментария из непосредственного суперкласса.

{@link}

Дескриптор {@link} предоставляет встроенную ссылку на дополнительную информацию. Его синтаксис выглядит так:

```
{@link mod-name/pkg-name.class-name#member-name text}
```

В mod-name/pkg-name.class-name#member-name указывается имя класса или метода, к которому добавляется ссылка, а в text — отображаемая строка.

Параметр text является необязательным. Если он не включен, то member-name отображается в виде ссылки. Обратите внимание, что имя модуля mod-name (если есть) отделено от имени пакета pkg-name символом /. Например, следующий дескриптор определяет ссылку на метод write() класса Writer из пакета java.io в модуле java.base:

```
{@link java.base/java.io.Writer#write}
```

{@linkplain}

Дескриптор {@linkplain} вставляет встроенную ссылку на другую тему. Ссылка отображается обычным текстовым шрифтом. В остальном он похож на {@link}.

{@literal}

Дескриптор {@literal} позволяет вставить текст в комментарий. Затем этот текст отображается в том виде как есть без какой-либо дальнейшей обработки вроде визуализации HTML. Вот его синтаксис:

```
{@literal description}
```

В description указывается вставляемый текст.

@param

Дескриптор @param документирует параметр. Он имеет такой синтаксис:

```
@param parameter-name explanation
```

В parameter-name указывается имя параметра, а его смысл описывается в explanation. Дескриптор @param можно применять только в документации для метода или конструктора либо обобщенного класса или интерфейса.

@provides

Дескриптор @provides документирует службу, предоставляемую модулем. Его синтаксис выглядит следующим образом:

```
@provides type explanation
```

В type указывается тип поставщика службы, а в explanation — его описание.

@return

Дескриптор @return описывает возвращаемое значение метода. Он имеет две формы. Первая — блочный дескриптор:

```
@return explanation
```

В explanation задается описание типа и смысла значения, возвращаемого методом. Таким образом, дескриптор @return можно использовать только в документации для метода.

В версии JDK 16 добавлена встроенная версия дескриптора:

```
{@return explanation}
```

Такая форма должна находиться в начале документирующего комментария для метода.

@see

Дескриптор @see предоставляет ссылку на дополнительную информацию. Ниже показаны две часто применяемые формы:

```
@see anchor
@see mod-name/pkg-name.class-name#member-name text
```

В первой форме anchor — это ссылка на абсолютный или относительный URL. Во второй форме в mod-name/pkg-name.class-name#member-name указывается имя элемента, а в text — текст, отображаемый для данного элемента. Параметр text является необязательным, и если он не используется, тогда отображается элемент, указанный в mod-name/pkg-name.class-name#member-name. Имя члена (member-name) тоже задавать необязательно. Таким образом, в дополнение к ссылке на конкретный метод или поле можно указывать ссылку на модуль, пакет, класс или интерфейс. Имя может быть полным или частичным. Тем не менее, символ точки, предшествующий имени члена (если есть), должен быть заменен символом решетки. Существует третья форма @see, которая позволяет просто указывать текстовое описание.

@serial

Дескриптор @serial определяет комментарий для стандартного сериализуемого поля. Вот его основная форма:

```
@serial description
```

В description задается описание для этого поля. Две другие формы позволяют указывать, будет ли класс или пакет частью страницы документации Serialized Form (Сериализированная форма):

```
@serial include
@serial exclude
```

@serialData

Дескриптор @serialData документирует данные, записанные методами writeObject() и writeExternal(). Он имеет следующий синтаксис:

```
@serialData description
```

В description указывается описание для таких данных.

@serialField

Для класса, реализующего интерфейс Serializable, дескриптор @serialField предоставляет комментарии для компонента ObjectStreamField.

Его синтаксис показан ниже:

```
@serialField name type description
```

В name задается имя поля, в type — его тип, а в description — описание поля.

@since

Дескриптор @since указывает, что элемент появился в конкретном выпуске. Вот его синтаксис:

```
@since release
```

В release задается строка, обозначающая выпуск или версию, в которой данное функциональное средство стало доступным.

{@summary}

Дескриптор {@summary} явно указывает текст сводки, который будет применяться с элементом. Он должен быть первым дескриптором в документации для элемента и имеет следующий синтаксис:

```
@summary explanation
```

В explanation приводится сводка по элементу, которая может занимать несколько строк. Если дескриптор {@summary} отсутствует, тогда в качестве сводки используется первая строка документирующего комментария для элемента.

{@systemProperty}

Дескриптор {@systemProperty} позволяет указывать системное свойство. Ниже представлен его общий вид:

```
{@systemProperty propName}
```

В propName указывается имя свойства.

@throws

Дескриптор @throws делает то же, что и @exception, но теперь является предпочтительной формой.

@uses

Дескриптор @uses документирует поставщика службы, необходимой модулю. Вот его синтаксис:

```
@uses type explanation
```

В type указывается тип поставщика службы, а в explanation — описание службы.

{@value}

Дескриптор {@value} имеет две формы. Первая отображает значение предшествующей константы, которая должна быть статическим полем:

```
{@value}
```

Вторая форма отображает значение заданного статического поля:

```
{@value pkg.class#field}
```

В pkg.class#field указывается имя статического поля.

@version

Дескриптор @version указывает версию элемента программы и имеет следующий синтаксис:

```
@version info
```

В info задается строка, содержащая информацию о версии, обычно номер версии, такой как 2.2. Чтобы информация в @version была включена в документацию HTML, при запуске javadoc потребуется указать параметр -version.

Общая форма документирующего комментария

После начальных символов /** первая строка или строки становятся главным описанием класса, интерфейса, поля, конструктора, метода или модуля. Затем можно добавить один или несколько дескрипторов @. Каждый дескриптор @ должен начинаться с новой строки или следовать за одной или несколькими звездочками (*) в начале строки. Дескрипторы одного типа должны группироваться вместе. Скажем, три дескриптора @see необходимо указывать друг за другом. Встроенные дескрипторы (начинающиеся с фигурной скобки) можно применять внутри любого описания.

Ниже представлен пример документирующего комментария для класса:

```
/**
  Это класс вычерчивает гистограмму.
  @author Herbert Schildt
  @version 3.2
*/
```

Вывод утилиты javadoc

Утилита javadoc принимает в качестве входных данных исходный файл программы на Java и выводит несколько файлов HTML, содержащих документацию по программе. Информация о каждом классе будет находиться в отдельном файле HTML. Утилита javadoc также выводит индекс и дерево иерархии. Могут быть сгенерированы и другие файлы HTML. Начиная с версии JDK 9, также включено поле поиска.

Пример использования документирующих комментариев

Далее приведен пример программы, в которой применяются документирующие комментарии. Обратите внимание, что каждый комментарий непосредственно предшествует элементу, который он описывает. После обработки утилитой javadoc документация по классу SquareNum будет находиться в файле SquareNum.html.

```
import java.io.*;
/**
  В этом классе демонстрируется использование документирующих комментариев.
  @author Herbert Schildt
  @version 1.2
*/
public class SquareNum {
/**
  Этот метод возвращает квадрат числа num.
  Описание содержит несколько строк.
  Количество строк может быть произвольным.
  @param num Значение, возводимое в квадрат.
  @return Квадрат числа num.
*/
  public double square(double num) {
    return num * num;
  }
  /**
    Этот метод вводит число от пользователя.
    @return Введенное значение типа double.
    @throws IOException Возникает в случае ошибки при вводе.
    @see IOException
  */
  public double getNumber() throws IOException {
    // Создать объект BufferedReader, использующий System.in.
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader inData = new BufferedReader(isr);
    String str;
    str = inData.readLine();
    return (new Double(str)).doubleValue();
  }
  /**
    Этот метод демонстрирует применение square().
    @param args Не используется.
    @throws IOException Возникает в случае ошибки при вводе.
    @see IOException
  */
  public static void main(String[] args) throws IOException
  {
    SquareNum ob = new SquareNum();
    double val;
    System.out.println(«Введите значение для возведения в квадрат: «);
    val = ob.getNumber();
    val = ob.square(val);
    System.out.println(«Возведенное в квадрат значение: « + val);
  }
}
```

ПРИЛОЖЕНИЕ

Б Введение в JShell

Начиная с версии JDK 9, в состав Java входит инструмент под названием JShell. Он предоставляет интерактивную среду, позволяющую быстро и легко экспериментировать с кодом Java. Инструмент JShell реализует механизм выполнения *цикла "чтение-вычисление-вывод"* (read-evaluate-print loop — REPL). Механизм REPL предлагает ввести фрагмент кода, который читается и вычисляется. Затем JShell отображает вывод, связанный с кодом, такой как вывод, произведенный оператором `println()`, результат выражения или текущее значение переменной. Далее инструмент JShell запрашивает следующий фрагмент кода, и процесс продолжается (т.е. организуется цикл). В контексте JShell каждая введенная последовательность кода называется *фрагментом*.

Ключевым моментом в понимании инструмента JShell является то, что для его использования не придется вводить полную программу на Java. Каждый введенный фрагмент просто вычисляется по мере его ввода. Это становится возможным, поскольку JShell автоматически обрабатывает многие детали, связанные с программой на Java, позволяя сосредоточить внимание на конкретной функции без необходимости писать полную программу, что делает JShell особенно удобным на начальной стадии изучения Java.

Как и следовало ожидать, инструмент JShell может быть полезен и для опытных программистов. Так как JShell хранит информацию о состоянии, можно вводить многострочные кодовые последовательности и запускать их внутри JShell. В результате инструмент JShell крайне удобен, когда нужно создать прототип концепции, потому что он позволяет интерактивно экспериментировать с кодом без необходимости разрабатывать и компилировать полную программу.

В настоящем приложении предлагается введение в JShell и исследование нескольких его основных возможностей с акцентом на средствах, наиболее полезных для начинающих программистов на Java.

Основы JShell

JShell представляет собой инструмент командной строки. Таким образом, он запускается в окне командной строки. Чтобы запустить сеанс JShell, пона-

добится ввести команду `jshell` в командной строке; появится приглашение на ввод JShell:

```
jshell>
```

После отображения приглашения можно вводить фрагмент кода или команду JShell. В своей простейшей форме JShell позволяет ввести отдельный оператор и сразу увидеть результат. Для начала возвратимся к самой первой программе на Java, приведенной в книге, которая снова показана ниже:

```
class Example {
  // Программа начинается с вызова main().
  public static void main(String[] args) {
    System.out.println("Простая программа на языке Java.");
  }
}
```

В программе только оператор `println()` фактически выполняет действие, связанное с отображением его сообщения на экране. Остальной код просто предоставляет необходимые объявления классов и методов. В JShell нет необходимости явно указывать класс или метод для выполнения оператора `println()`. Инструмент JShell способен выполнить его самостоятельно. Введите следующую строку в командной строке JShell:

```
System.out.println("Простая программа на языке Java.");
```

Затем нажмите <Enter>. Отобразится вывод:

```
Простая программа на языке Java.

jshell>
```

Как видите, метод `println()` вызывается и выводит свой строковый аргумент. Далее снова отображается приглашение.

Прежде чем двигаться дальше, полезно выяснить, почему инструмент JShell может выполнить одиночный оператор, скажем, вызов `println()`, тогда как компилятору `javac` языка Java требуется полная программа. Инструмент JShell способен выполнять один оператор из-за автоматического обеспечения "за кулисами" необходимой программной инфраструктуры, которая состоит из *синтетического класса* и *синтетического метода*. Таким образом, в данном случае оператор `println()` встраивается в синтетический метод, входящий в состав синтетического класса. В результате приведенный выше код по-прежнему является частью корректной программы на Java, даже если все детали не видны. Такой подход предлагает очень быстрый и удобный способ экспериментирования с кодом Java.

Далее взглянем, как поддерживаются переменные. В JShell можно объявлять переменную, присваивать ей значение и задействовать ее в любых допустимых выражениях. Например, введите в командной строке JShell такую строку:

```
int count;
```

Вы увидите следующий ответ:

```
count ==> 0
```

Ответ указывает на то, что переменная count была добавлена в синтетический класс и инициализирована нулем. Более того, она была добавлена как статическая переменная синтетического класса.

Затем присвойте count значение 10 с помощью показанного ниже оператора:

```
count = 10;
```

Вот как выглядит ответ:

```
count ==> 10
```

Легко заметить, что count теперь имеет значение 10. Из-за того, что переменная count статическая, ее можно применять без ссылки на объект.

Теперь, когда переменная count объявлена, ее можно использовать в выражении. Например, введите следующий оператор println():

```
System.out.println("Обратная величина count: " + 1.0 / count);
```

Вот как отреагирует JShell:

```
Обратная величина count: 0.1
```

Результат выражения 1.0 / count равен 0.1, потому что переменной count ранее было присвоено значение 10.

Помимо применения переменной в предыдущем примере демонстрируется еще один важный аспект инструмента JShell: поддержка информации о состоянии. В данном случае переменной count присваивается значение 10 в одном операторе, после чего это значение используется в выражении 1.0 / count внутри вызова println() во втором операторе. Инструмент JShell сохраняет значение count между двумя введенными операторами. Как правило, JShell поддерживает текущее состояние и действие вводимых фрагментов кода, что позволяет экспериментировать с более крупными фрагментами кода, занимающими несколько строк.

Рассмотрим еще один пример. Здесь будет создан цикл for, в котором применяется переменная count. Для начала понадобится ввести такую строку:

```
for(count = 0; count < 5; count++)
```

Инструмент JShell отреагирует следующим приглашением:

```
...>
```

Оно указывает на то, что для завершения оператора требуется дополнительный код. В данном случае необходимо указать цель цикла for:

```
System.out.println(count);
```

После ввода этой строки оператор for становится завершенным, и выполняются обе строки. Вы увидите показанный ниже вывод:

```
0
1
2
3
4
```

В дополнение к операторам и объявлениям переменных инструмент JShell позволяет объявлять классы и методы, а также использовать операторы `import`. Примеры приводятся в последующих разделах. И еще один момент: любой код, допустимый для JShell, также будет допустимым для компиляции с помощью `javac` при условии, что для создания полной программы обеспечена вся необходимая структура. Таким образом, если фрагмент кода может быть выполнен в JShell, то он является действительным кодом Java. Другими словами, код JShell — это код Java.

Просмотр, редактирование и повторного выполнение кода

Инструмент JShell поддерживает большое количество команд, позволяющих управлять его работой. На текущем этапе особый интерес представляют три из них, поскольку они позволяют просматривать введенный код, редактировать строку кода и повторно запускать фрагмент кода. Когда последующие примеры станут длиннее, вы оцените полезность таких команд.

Все команды JShell начинаются с символа /, за которым следует сама команда. Возможно, наиболее часто применяемой командой считается `/list`, которая позволяет просматривать введенный код. Предполагая, что вы следовали примерам, приведенным в предыдущем разделе, можете просмотреть код, введя команду `/list`. Сеанс JShell отреагирует выдачей пронумерованного списка ранее введенных фрагментов. Обратите особое внимание на фрагмент с циклом `for`. Несмотря на наличие двух строк, он представляет собой один оператор, и потому используется только один номер фрагмента. На языке JShell номера фрагментов называются *идентификаторами фрагментов*. Помимо только что показанной базовой формы `/list` поддерживаются и другие формы, в том числе те, которые позволяют перечислять конкретные фрагменты по именам или номерам. Например, вывести объявление `count` можно с применением команды `/list count`.

Для редактирования фрагмента предназначена команда `/edit`, которая приводит к открытию окна редактирования, где можно модифицировать код. Сейчас пригодятся следующие три формы команды `/edit`. Во-первых, если указать просто `/edit`, то окно редактирования будет содержать все введенные ранее строки и позволит редактировать любую их часть. Во-вторых, с использованием `/edit n`, где `n` — номер фрагмента, можно указать конкретный фрагмент для редактирования. Например, чтобы отредактировать фрагмент номер 3, понадобится ввести `/edit 3`. В-третьих, можно указывать именованный элемент, такой как переменная. Скажем, для изменения значения `count` нужно применить `/edit count`.

Вы уже видели, что JShell выполняет код по мере его ввода. Однако можно также повторно запускать то, что было введено ранее. Для повторного запуска последнего введенного фрагмента используется `/!`. Чтобы повторно запустить определенный фрагмент, необходимо указать его номер в форме `/n`,

где n — номер фрагмента, подлежащего запуску. Например, для повторного запуска четвертого фрагмента нужно ввести /4. Повторно запустить фрагмент можно и за счет указания его позиции относительно текущего фрагмента с применением отрицательного смещения. Например, чтобы повторно запустить фрагмент, который находится на три фрагмента раньше текущего, понадобится использовать /-3.

В настоящий момент важно отметить, что несколько команд, в том числе только что показанные, позволяют задавать список имен или номеров. Скажем, для редактирования строк 2 и 4 можно применить команду /edit 2 4. В последних версиях JShell несколько команд позволяют указывать диапазон фрагментов. К ним относятся только что описанные команды /list, /edit и /n. Например, чтобы перечислить фрагменты с 4 по 6, необходимо использовать команду /list 4-6.

Есть еще одна команда, о которой следует знать: /exit. Она заканчивает работу JShell.

Добавление метода

Как объяснялось в главе 6, методы находятся внутри классов. Тем не менее, в JShell можно экспериментировать с методом, не объявляя его *явно* в классе. Как упоминалось ранее, дело в том, что JShell автоматически помещает фрагменты кода внутрь синтетического класса. В результате можно легко и быстро написать метод, не предоставляя инфраструктуру классов. Кроме того, метод можно вызывать, не создавая объект. Такая возможность JShell особенно полезна при изучении основ методов в Java или при написании прототипов для нового кода. Чтобы лучше понять процесс, рассмотрим пример.

Первым делом запустите новый сеанс JShell и введите следующий фрагмент после появления приглашения:

```
double reciprocal(double val) {
  return 1.0/val;
}
```

Фрагмент создает метод, который возвращает обратную величину своего аргумента. Вот как JShell отреагирует:

```
|      created method reciprocal(double)
|      создан метод reciprocal(double)
```

Ответ указывает на то, что метод был добавлен в синтетический класс JShell и готов к применению.

Чтобы вызвать метод reciprocal(), нужно просто указать его имя без какой-либо ссылки на объект или класс, например:

```
System.out.println(reciprocal(4.0));
```

Инструмент JShell отобразит значение 0.25.

Вас может интересовать, по какой причине метод reciprocal() можно вызывать, не используя операцию точки и ссылку на объект. И вот ответ.

Когда создается автономный метод в JShell, такой как `reciprocal()`, он автоматически делается статическим членом синтетического класса. Как объяснялось в главе 7, статические методы вызываются относительно их класса, а не конкретного объекта. Таким образом, объект не требуется. Ситуация похожа на то, как автономные переменные становятся статическими переменными синтетического класса.

Еще одним важным аспектом JShell является поддержка *опережающей ссылки* внутри метода. Такое средство позволяет одному методу вызывать другой метод, даже если тот еще не определен, что дает возможность вводить метод, зависящий от другого метода, и не беспокоиться о том, какой из методов вводится первым. Рассмотрим простой пример. Введите в JShell приведенную ниже строку:

```
void myMeth() { myMeth2(); }
```

Инструмент JShell отреагирует следующим образом:

```
|  created method myMeth(), however, it cannot be invoked until myMeth2()
is declared
|  создан метод myMeth(), однако он не может быть вызван до объявления
myMeth2()
```

Как видите, инструменту JShell известно, что метод `myMeth2()` еще не объявлен, но он все-таки позволяет определить `myMeth()`. Вполне ожидаемо попытка вызова `myMeth()` в данный момент приведет к выдаче сообщения об ошибке, поскольку `myMeth2()` еще не определен, но код для `myMeth()` вводить разрешено.

Далее определите метод `myMeth2()`:

```
void myMeth2() { System.out.println("JShell - эффективный инструмент."); }
```

Теперь метод `myMeth2()` определен и можно вызывать `myMeth()`.

Помимо методов опережающую ссылку можно применять в инициализаторе поля внутри класса.

Создание класса

Хотя JShell автоматически предоставляет синтетический класс, внутрь которого помещаются фрагменты кода, в JShell можно также создавать собственный класс. Вдобавок разрешено создавать экземпляры такого класса. В результате появляется возможность экспериментирования с классами в интерактивной среде JShell. Процесс иллюстрируется в следующем примере.

Запустите новый сеанс JShell и введите следующий код класса:

```
class MyClass {
  double v;

  MyClass(double d) { v = d; }

  // Возвратить обратную величину v.
  double reciprocal() { return 1.0 / v; }
}
```

После завершения ввода JShell отреагирует, как показано ниже:

```
|  created class MyClass
|  создан класс MyClass
```

Теперь, когда класс `MyClass` добавлен, им можно пользоваться. Скажем, вот как создать объект `MyClass`:

```
MyClass ob = new MyClass(10.0);
```

Инструмент JShell сообщит о добавлении `ob` в качестве переменной типа `MyClass`. Затем введите следующую строку:

```
System.out.println(ob.reciprocal());
```

Инструмент JShell отреагирует отображением значения `0.1`.

Интересно отметить, что добавляемый в JShell класс становится статическим вложенным членом синтетического класса.

Использование интерфейса

Интерфейсы поддерживаются JShell аналогично классам. Таким образом, в JShell можно объявлять интерфейс и реализовывать его посредством класса. Рассмотрим простой пример. Первым делом запустите новый сеанс JShell.

В интерфейсе, который будет применяться, объявлен метод `isLegalVal()`, предназначенный для определения допустимости значения для какой-либо цели. Он возвращает `true`, если значение допустимо, или `false` в противном случае. Разумеется, то, что представляет собой допустимое значение, будет определяться каждым классом, реализующим интерфейс. Начните с ввода в JShell приведенного далее кода интерфейса:

```
interface MyIF {
  boolean isLegalVal(double v);
}
```

Инструмент JShell отреагирует следующим образом:

```
|  created interface MyIf
|  создан интерфейс MyIf
```

Затем введите код класса, который реализует интерфейс `MyIF`:

```
class MyClass implements MyIF {

  double start;
  double end;

  MyClass(double a, double b) { start = a; end = b; }

  // Выяснить, находится ли значение v внутри диапазона
  // от start до end включительно.
  public boolean isLegalVal(double v) {
    if((v >= start) && (v <= end)) return true;
    return false;
  }

}
```

Вот как отреагирует инструмент JShell:

```
|  created class MyClass
|  создан класс MyClass
```

Обратите внимание, что в MyClass реализован метод isLegalVal(), который выясняет, находится ли значение v внутри диапазона значений, заданного в переменных экземпляра start и end класса MyClass включительно.

После добавления MyIF и MyClass можно создать объект MyClass и вызвать на нем метод isLegalVal():

```
MyClass ob = new MyClass(0.0, 10.0);
System.out.println(ob.isLegalVal(5.0));
```

В данном случае отображается true, потому что значение 5 находится внутри диапазона от 0 до 10.

Поскольку интерфейс MyIF был добавлен в JShell, можно также создавать ссылку типа MyIF. Например, следующий код является допустимым:

```
MyIF ob2 = new MyClass(1.0, 3.0);
boolean result = ob2.isLegalVal(1.1);
```

В этом случае значением result будет true, о чем и сообщит инструмент JShell. И еще один момент: аналогично классам и интерфейсам в JShell поддерживаются перечисления и аннотации.

Вычисление выражений и использование встроенных переменных

Инструмент JShell обладает возможностью непосредственного вычисления выражений без необходимости их включения в полный оператор Java, что особенно полезно при экспериментировании с кодом и отсутствии необходимости выполнять выражение в более широком контексте. Рассмотрим простой пример. Запустив новый сеанс JShell, введите в командной строке следующее выражение:

```
3.0 / 16.0
```

JShell выдаст:

```
$1 ==> 0.1875
```

Как видите, результат выражения вычисляется и отображается. Однако обратите внимание, что результирующее значение также присваивается временной переменной по имени $1. В общем случае при вычислении выражения напрямую его результат сохраняется во временной переменной подходящего типа. Все имена временных переменных начинаются с символа $, за которым следует число, увеличивающееся каждый раз, когда требуется новая временная переменная. Временными переменными можно пользоваться точно так же, как и любыми другими переменными. Скажем, в показанном ниже примере отображается значение переменной $1, которое в данном случае равно 0.1875:

```
System.out.println($1);
```

Вот еще один пример:

```
double v = $1 * 2;
```

Здесь v присваивается значение $1, умноженное на 2. Таким образом, v будет содержать 0.375.

Значение временной переменной можно изменять. Например, следующий код меняет знак $1 на противоположный:

```
$1 = -$1
```

JShell выдаст:

```
$1 ==> -0.1875
```

Выражения не ограничиваются числовыми значениями. Вот пример, в котором выполняется конкатенация строки со значением, возвращаемым Math.abs($1):

```
"Абсолютное значение $1 равно " + Math.abs($1)
```

Результатом будет временная переменная, содержащая строку:

```
Абсолютное значение $1 равно 0.1875
```

Импортирование пакетов

Как описано в главе 9, оператор import применяется для того, чтобы сделать доступными элементы пакета. Кроме того, каждый раз, когда используется пакет, отличающийся от java.lang, он должен быть импортирован. Ситуация с инструментом JShell почти такая же, за исключением того, что по умолчанию автоматически импортируется ряд часто применяемых пакетов. К ним относятся помимо прочих java.io и java.util. Поскольку эти пакеты уже импортированы, явный оператор import для их использования не требуется.

Скажем, по причине автоматического импортирования пакета java.io можно ввести следующий оператор:

```
FileInputStream fin = new FileInputStream("myfile.txt");
```

Вспомните, что класс FileInputStream находится в пакете java.io. Из-за того, что java.io импортируется автоматически, с ним можно работать без явного оператора import. Предполагая наличие в текущем каталоге файла по имени myfile.txt, инструмент JShell отреагирует добавлением переменной fin и открытием файла. Далее содержимое файла можно прочитать и отобразить с помощью таких операторов:

```
int i;
do {
  i = fin.read();
  if(i != -1) System.out.print((char) i);
} while(i != -1);
```

Это тот же самый базовый код, который обсуждался в главе 13, но явный оператор import java.io здесь не нужен.

Имейте в виду, что JShell автоматически импортирует лишь небольшое количество пакетов. Чтобы использовать пакет, который не импортируется JShell автоматически, его потребуется явно импортировать, как делается в нормальной программе на Java. Кроме того, просмотреть текущий список импортированных пакетов можно посредством команды `/imports`.

Исключения

В примере с вводом-выводом, приведенном в предыдущем разделе, фрагменты кода также иллюстрируют еще один очень важный аспект JShell. Обратите внимание, что блоки `try/catch` для обработки исключений ввода-вывода отсутствуют. Взглянув на аналогичный код в главе 13, вы увидите, что код, открывающий файл, перехватывает исключение `FileNotFoundException`, а код, читающий файл, отслеживает исключение `IOException`. Причина, по которой не нужно перехватывать эти исключения в показанных ранее фрагментах, связана с тем, что JShell обрабатывает их автоматически. В более общем смысле JShell во многих случаях автоматически обрабатывает проверяемые исключения.

Другие команды JShell

Кроме рассмотренных ранее команд инструмент JShell поддерживает несколько других. Одной из команд, которую имеет смысл опробовать незамедлительно, является `/help`. Она отображает список команд. Для получения справочной информации можно также применять команду `/?`. Ниже кратко описаны некоторые наиболее часто используемые команды.

Команда `/reset` позволяет сбросить JShell в начальное состояние, что особенно полезно при переходе к новому проекту. За счет применения `/reset` устраняется необходимость выхода и перезапуска JShell. Тем не менее, имейте в виду, что команда `/reset` сбрасывает всю среду JShell, а потому вся информация о состоянии утрачивается.

Сохранить сеанс можно с помощью команды `/save`. Вот ее простейшая форма:

```
/save filename
```

В `filename` указывается имя файла для сохранения сеанса. По умолчанию `/save` сохраняет текущий исходный код, но поддерживает несколько параметров, два из которых представляют особый интерес. Параметр `-all` позволяет сохранить все введенные ранее строки, в том числе и те, которые были введены некорректно. Параметр `-history` дает возможность сохранить хронологию сеанса (т.е. список введенных команд).

Загрузить сохраненный сеанс можно с использованием команды `/open`, которая имеет следующую форму:

```
/open filename
```

В `filename` указывается имя файла для загрузки сеанса.

Инструмент JShell предоставляет несколько команд, которые позволяют посматривать различные элементы проделанной работы. Они показаны в табл. Б.1.

Таблица Б.1. Команды для просмотра различных элементов

Команда	Описание
/types	Отображает классы, интерфейсы и перечисления
/imports	Отображает операторы import
/methods	Отображает методы
/vars	Отображает переменные

Например, в случае ввода приведенных далее строк:

```
int start = 0;
int end = 10;
int count = 5;
```

и последующего ввода команды /vars вы увидите такой вывод:

```
|  int start = 0;
|  int end = 10;
|  int count = 5;
```

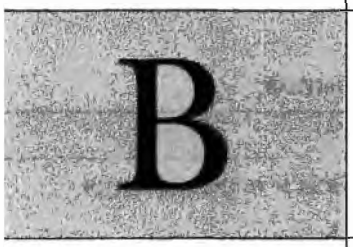
Еще одна часто применяемая команда — /history. Она позволяет просматривать хронологию текущего сеанса, которая содержит список всего, что вводилось в командной строке.

Дальнейшее исследование JShell

Осваивать инструмент JShell эффективнее всего за счет работы с ним. Попробуйте ввести несколько конструкций Java и посмотрите, как реагирует JShell. В ходе экспериментов с JShell вы обнаружите шаблоны использования, которые подходят вам лучше других. Такой подход позволит отыскать эффективные способы интеграции JShell в ваш процесс обучения или разработки. Кроме того, имейте в виду, что инструмент JShell предназначен не только для новичков. Он также превосходен при прототипировании кода. Таким образом, даже опытные профессионалы сочтут JShell полезным, когда им нужно исследовать новые области.

Подводя итоги, можно утверждать, что JShell является важным инструментом, который еще больше расширяет возможности разработки на языке Java.

ПРИЛОЖЕНИЕ



Компиляция и запуск простых однофайловых программ за один шаг

В главе 2 было показано, как компилировать программу на Java в байт-код с помощью компилятора `javac` и затем запустить полученный файл или файлы `.class` с использованием средства запуска `java`. Подобным образом программы на Java компилировались и запускались с самого начала существования Java, и именно такой способ вы будете использовать при разработке приложений. Однако, начиная с версии JDK 11, некоторые виды простых программ на Java можно компилировать и запускать непосредственно из исходного файла без предварительного вызова `javac`. Для этого средству запуска `java` понадобится передать имя файла исходного кода с расширением `.java`, что заставит `java` автоматически вызвать компилятор и выполнить программу.

Скажем, следующая команда приводит к автоматической компиляции и запуску первого примера программы, приведенного в книге:

```
java Example.java
```

В данном случае класс `Example` компилируется и запускается за один шаг. Применять `javac` не нужно. Тем не менее, имейте в виду, что файл `.class` не создается, а компиляция выполняется "за кулисами". В результате для повторного запуска программы придется снова запустить файл исходного кода. Выполнить его файл `.class` не удастся, т.к. он не создается.

Возможность запуска файла исходного кода облегчает использование программ на Java в файлах сценариев. Она также может быть полезна для коротких одноразовых программ. В некоторых случаях такая возможность упрощает запуск простых программ-примеров при экспериментировании с Java. Однако это не универсальная замена нормального процесса компиляции и запуска программ на Java.

Несмотря на привлекательность новой возможности запуска программы на Java прямо из файла исходного кода, с ней связан ряд ограничений. Во-первых, вся программа должна содержаться в одном файле исходного кода, но большинство реальных программ располагаются в нескольких файлах. Во-вторых, всегда будет выполняться первый найденный в файле класс, который должен содержать метод `main()`. Если первый класс в файле не содер-

жит `main()`, тогда запуск завершится ошибкой. Это означает необходимость соблюдения строгой организации кода, даже если вы предпочитаете организовывать его иначе. В-третьих, поскольку файлы `.class` не создаются, применение `java` для запуска однофайловой программы не приводит к созданию файла класса, который можно многократно использовать, возможно, в других программах. Из-за указанных ограничений применение `java` для запуска однофайловой программы может быть полезным, но по существу представляет собой методику особого случая.

Что касается этой книги, то средство запуска однофайловой программы имеет смысл применять при проработке многих примеров; просто удостоверьтесь, что класс с методом `main()` находится первым в файле. Тем не менее, данный прием не является применимым или уместным во всех случаях. Кроме того, в обсуждениях и многих примерах, приводимых в книге, предполагается использование нормального процесса компиляции, когда с применением `javac` файл исходного кода компилируется в байт-код, а для запуска результирующего байт-кода используется `java`. Такой механизм применяется при реальной разработке и потому его понимание — важная часть изучения Java. Крайне важно иметь хорошее представление о нем. По указанным причинам при работе с примерами настоятельно рекомендуется во всех случаях использовать обычный подход к компиляции и запуску программ на Java, что гарантирует наличие прочных знаний работы Java. Разумеется, вы наверняка захотите поэкспериментировать с вариантом запуска однофайловых программ.

На заметку! Однофайловую программу можно выполнить из файла, не имеющего расширение `.java`. Для этого понадобится указать параметр `--source APIVer`, где `APIVer` — номер версии JDK.

Предметный указатель

Л