ORACLE *Oracle Press*™

The Complete Reference

Herbert Schildt

New York Chicago San Francisco
Lisbon London Madrid Mexico City
Milan New Delhi San Juan
Seoul Singapore Sydney Toronto

Полное руководство

Герберт Шилдт

Москва • Санкт-Петербург • Киев
2012

ББК 32.973.26-018.2.75
Ш57
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция *В.А. Коваленко*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Шилдт, Герберт.
Ш57 Java. Полное руководство, 8-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2012. — 1104 с. : ил. — Парал. тит. англ.
ISBN 978-5-8459-1759-1 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Герберт Шилдт

Java. Полное руководство

8-е издание

Литературный редактор *Е.Д. Давидян*
Верстка *О.В. Мишутина*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 10.02.2012. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 89,01. Уч.-изд. л. 62,0.
Тираж 1500 экз. Заказ № 3030.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9-я линия В. О., д. 12.

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1759-1 (рус.)
ISBN 978-0-07-160630-1 (англ.)

Оглавление

Содержание

Об авторе

Герберт Шилдт – известный во всем мире автор множества книг, посвященных программированию на языках Java, C++, C и C#. Его книги продаются миллионными тиражами и переводятся на множество языков мира. К успешным книгам Герберта по языку Java относятся *Java: руководство для начинающих*; *Java: методики программирования Шилдта*; *SWING: руководство для начинающих*; и *Искусство программирования на Java*. Бестселлерами по C++ являются *Полный справочник по C++*; *C# 4: полное руководство*; и *C: полное руководство, классическое издание*. Интересуясь всеми компьютерными аспектами, он уделяет основное внимание языкам программирования, включая компиляторы, интерпретаторы и языки управления роботами. Он также проявляет активный интерес к стандартизации языков. Герберт имеет диплом о высшем образовании, а также ученую степень, которую получил в университете Иллинойса. Дополнительная информация об авторе представлена на его веб-сайте по адресу: `www.HerbSchildt.com`.

О техническом редакторе

Д-р Дэни Ковард (Danny Coward) является соавтором по платформам Java Platforms с 1997 года. Он был членом-учредителем группы Java EE со времен корпорации Sun, членом исполнительного комитета Java Community Process Executive Committee, ведущим соавтором всех выпусков платформ Java Platform (Java SE, Java ME и Java EE), а также основателем первоначальной группы JavaFX.

Введение

Java — один из самых важных и популярных компьютерных языков в мире. Кроме того, он удерживает свое лидерство на протяжении многих лет. В отличие от некоторых других языков программирования, влияние которых со временем уменьшилось, язык Java стал только сильнее. Со времен своего первого выпуска он выдвинулся на передний край программирования для Интернета. Каждая последующая версия укрепляла эту позицию. Сегодня он все еще первый и является наилучшим выбором для разработки веб-ориентированных приложений. Благодаря языку Java стала также возможной современная революция смартфонов, поскольку он используется для написания программ для платформы Android. Просто представьте: большая часть современного кода в мире является кодом Java. Язык Java действительно столь важен.

Основная причина успеха языка Java — его быстрая изменчивость. Начиная с его первого выпуска 1.0 этот язык непрерывно адаптируется к изменениям в среде программирования и подходов к программированию. Самое главное — он не просто следует тенденциям, *а помогает их создавать*. Способность адаптации языка Java к высокой скорости изменений в компьютерном мире — основная причина, по которой он остается столь успешным языком.

Со времени публикации первого издания этой книги в 1996 году, она претерпела множество изменений, которые отражали последовательное развитие языка Java. Это, восьмое, издание адаптировано для Java SE 7, поэтому оно содержит значительный объем нового материала. Например, сюда включено описание расширений языка Project Coin, дополнительных средств NIO (NIO.2) и Fork/Join Framework. Как правило, описание новых средств интегрировано в существующие главы, однако, в связи с большим объемом добавлений в NIO, эта тема теперь представлена в отдельной главе. Тем не менее общая структура книги остается такой же. Это значит, что если вы знакомы с предыдущим изданием, будете чувствовать себя комфортно и здесь.

Книга для всех программистов

Эта книга предназначена для всех программистов — как для новичков, так и для профессионалов. Начинающий программист найдет в ней подробные пошаговые описания и множество чрезвычайно полезных примеров, а углубленное рассмотрение более сложных функций и библиотек Java должно удовлетворить ожидания профессиональных программистов. Для обеих категорий читателей в книге указаны действующие ресурсы и полезные ссылки.

Что внутри

Эта книга представляет собой всеобъемлющее руководство по языку Java, описывающее его синтаксис, ключевые слова и фундаментальные принципы программирования. Здесь рассмотрена также значительная часть библиотеки Java API. Книга разделена на четыре части, каждая из которых посвящена отдельному аспекту среды программирования Java.

Часть I представляет собой подробный учебник по языку Java. Она начинается с рассмотрения основных понятий, таких как типы данных, операторы, управляющие операторы и классы. Затем описываются наследование, пакеты, интерфейсы, обработка исключений и многопоточное программирование. В заключительных главах этой части рассматриваются аннотации, перечисления, автоупаковка и обобщения.

В части II описаны основные аспекты стандартной библиотеки интерфейса прикладного программирования Java. В ней раскрыты такие темы, как строки, ввод-вывод, сетевая обработка, стандартные утилиты, инфраструктура коллекций Collections Framework, аплеты, элементы управления графического интерфейса пользователя (GUI), средства работы с изображениями и параллельной обработки (включая Fork/Join Framework).

В части III рассмотрены три важные технологии Java: Java Beans, сервлеты и Swing.

Часть IV содержит главы с примерами реального использования Java. Первая глава этой части посвящена разработке нескольких аплетов, которые выполняют популярные финансовые вычисления, такие как вычисление выплаты процентов по ссуде или размера минимального вклада, необходимого для получения желаемого ежемесячного дохода. В этой главе также показано, как преобразовать эти аплеты в сервлеты. В следующей главе этой части описана разработка утилиты загрузки, управляющей загрузкой файлов. К числу выполняемых им функций относятся запуск, останов и возобновление передачи данных. Обе главы заимствованы из моей книги *Искусство программирования на Java* (И.Д. "Вильямс", 2005), которую я написал в соавторстве с Джеймсом Холмсом (James Holmes).

Коды примеров доступны в веб

Помните, что исходные коды всех примеров, приведенных в этой книге, доступны на веб-сайте издательства по адресу: `www.oraclepressbooks.com`.

Особые благодарности

Выражаю особую благодарность Патрику Нотону (Patrick Naughton), Джо О'Нилу (Joe O'Neil), Джеймсу Холмсу (James Holms) и Дэни Коварду (Danny Coward).

Патрик Нотон был одним из создателей языка Java. Он также помог мне в написании первого издания этой книги. Значительная часть материала глав 19, 21 и 26 была предоставлена Патриком. Его проницательность, опыт и энергия в огромной степени способствовали успеху этой книги.

При подготовке второго и третьего изданий этой книги Джо О'Нил предоставил исходные черновые материалы, которые ныне можно найти в главах 28-29, 31-32. Джо помогал мне при написании нескольких книг, и я высоко ценю его вклад.

И наконец, я горячо благодарю Джеймса Холмса за подготовку материалов главы 34. Джеймс — выдающийся программист и автор. Он был моим соавтором

книги *Искусство программирования на Java*, автором книги *Struts: The Complete Reference* и соавтором книги *JSF: The Complete Reference.*

Герберт Шилдт

Для дальнейшего изучения

Эта книга открывает серию книг по программированию, написанных Гербертом Шилдтом. Ниже перечислены другие книги этого автора, которые, несомненно, вас заинтересуют.

Если хотите больше узнать о программировании на Java, рекомендую прочесть следующие книги.

- *Java: методики программирования Шилдта.* И.Д. "Вильямс", 2008 г.
- *Java: руководство для начинающих*, 5-е изд. И.Д. "Вильямс", 2012 г.
- *SWING: руководство для начинающих.* И.Д. "Вильямс", 2007 г.
- *Искусство программирования на Java.* И.Д. "Вильямс", 2005 г.

Тем, кто желает изучить язык C++, особенно полезными будут такие книги.

- *Полный справочник по C++*, 4-е изд. И.Д. "Вильямс", 2011 г.
- *C++: методики программирования Шилдта.* И.Д. "Вильямс", 2009 г.
- *C++: руководство для начинающих*, 2-е изд. И.Д. "Вильямс", 2005 г.
- *C++: базовый курс*, 3-е изд. И.Д. "Вильямс", 2011 г.
- *STL Programming From the Ground Up.* McGraw-Hill/Osborne

Для изучения C# рекомендуем следующие книги Шилдта.

- *C# 4.0: полное руководство.* И.Д. "Вильямс", 2012 г.
- *C# 3.0: руководство для начинающих, 2-е изд.* И.Д. "Вильямс", 2009 г.

Перечисленные ниже книги помогут в изучении языка C.

- *C: полное руководство, классическое издание.* И.Д. "Вильямс", 2010 г.

Если вам нужно быстро получить исчерпывающие ответы, обратитесь к книгам Герберта Шилдта — признанного во всем мире авторитета в области программирования.

Соглашения, принятые в этой книге

При оформлении книги были использованы соглашения, общепринятые в компьютерной литературе.

- Новые термины в тексте выделяются *курсивом.* Чтобы обратить внимание читателя на отдельные фрагменты текста, также применяется *курсив.*
- Текст программ, функций, переменных, URL, веб-страниц и другой код представлены моноширинным шрифтом.
- Все, что придется вводить с клавиатуры, выделено **полужирным моноширинным** шрифтом.

- Знакоместо в описаниях синтаксиса выделено *курсивом*. Это указывает на необходимость заменить его фактическим именем переменной, параметром или другим элементом, который должен находиться на этом месте `BINDSIZE=(`*`максимальная ширина колонки`*`)*(`*`номер колонки`*`)`.
- Пункты меню и названия диалоговых окон представлены следующим образом: Menu Option (Пункт меню).
- Разрыв слишком длинных строк кода, не помещающихся на странице, обозначен специальным символом ↳.

От издательства

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 интересно услышать и любые другие замечания, которые вам хотелось бы высказать авторам.

Мы ждем ваших комментариев. Вы можете прислать письмо по электронной почте или просто посетить наш веб-сервер, оставив на нем свои замечания. 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 подходящими для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш e-mail. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию следующих книг. Наши координаты:

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

Адреса для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1
Украины: 03150, Киев, а/я 152

ЧАСТЬ I

Язык Java

ГЛАВА 1

История и развитие языка Java

Чтобы досконально изучить язык Java, необходимо понять причины, которые привели к его созданию, факторы, повлиявшие на конечную архитектуру, и унаследованные им особенности. Подобно удачным компьютерным языкам, появившимся до него, Java объединяет в себе лучшие элементы из своего богатого наследия и новаторские концепции, применение которых обусловлено его уникальным положением. В то время как остальные главы этой книги посвящены практическим аспектам Java — в том числе его синтаксису, библиотекам и приложениям, — в настоящей главе описано, как и почему был разработан этот язык, что делает его столь важным и как он развился за годы своего существования.

Хотя язык Java неразрывно связан с Интернетом, важно помнить, что, прежде всего, это язык программирования. Модернизация и разработка компьютерных языков обусловлены двумя основными причинами:

- необходимостью адаптации к изменяющимся средам и областям применения;
- необходимостью реализации улучшений и усовершенствований в области программирования.

Как будет показано в этой главе, разработка языка Java почти в равной мере была обусловлена обеими этими причинами.

Происхождение языка Java

Язык Java тесно связан с языком C++, который, в свою очередь, является прямым наследником языка C. Многие особенности языка Java унаследованы от обоих этих языков. От C язык Java унаследовал свой синтаксис, а многие его объектно-ориентированные свойства были перенесены из C++. Собственно говоря, ряд определяющих характеристик Java был перенесен — или разработан в ответ на возникшие потребности — из его предшественников. Более того, создание языка Java своими корнями уходит глубоко в процесс усовершенствования и адаптации, который происходит в языках компьютерного программирования на протяжении нескольких последних десятилетий. Поэтому в данном разделе мы рассмотрим последовательность событий и факторы, которые привели к появлению языка Java. Как вы убедитесь, каждое новшество в архитектуре языка было обусловлено необходимостью решения той или иной фундаментальной проблемы, которую не могли решить существовавшие до этого языки. И Java — не исключение в этом отношении.

Зарождение современного программирования: язык C

Язык C буквально потряс компьютерный мир. Его влияние нельзя недооценивать, поскольку он полностью изменил подход к программированию. Создание языка C было прямым следствием потребности в структурированном, эффективном и высокоуровневом языке, который мог бы заменить код ассемблера в процессе создания системных программ. Как вы, вероятно, знаете, при проектировании компьютерного языка часто приходится находить компромиссы

- между простотой использования и предоставляемыми возможностями;
- между безопасностью и эффективностью;
- между устойчивостью и расширяемостью.

До появления языка C программистам, как правило, приходилось выбирать между языками, которые позволяли оптимизировать тот или иной набор характеристик. Например, хотя FORTRAN можно было использовать при написании достаточно эффективных программ для научных приложений, он не очень подходил для создания системного кода. Аналогично, в то время как язык BASIC был очень прост в изучении, он предоставлял не очень много функциональных возможностей, а его недостаточная структурированность ставила под сомнение его полезность при создании крупных программ. Язык ассемблера можно использовать для создания очень эффективных программ, но его трудно изучать и эффективно использовать. Более того, отладка ассемблерного кода может оказаться весьма сложной задачей.

Еще одной сыгравшей свою роль проблемой было то, что ранние языки программирования, такие как BASIC, COBOL и FORTRAN, были спроектированы без учета принципов структурирования. Вместо этого в них основными средствами управления программой были операторы безусловного перехода `GOTO`. В результате программы, созданные с применением этих языков, тяготели к созданию так называемого "макаронного кода" — множества запутанных переходов и условных ветвей, которые делали программу буквально недоступной для понимания. Хотя такие языки как Pascal и структурированы, они не были предназначены для создания максимально эффективных программ и были лишены ряда важных функций, необходимых для применения этих языков к написанию широкого круга программ. (В частности, учитывая существование нескольких стандартных диалектов Pascal, было нецелесообразно применять этот язык для создания кода системного уровня.)

Таким образом, непосредственно накануне изобретения языка C, несмотря на затраченные усилия, ни одному языку не удалось решить существующие конфликты. Вместе с тем потребность в таком языке становилась все более насущной. В начале 70-х гг. началась компьютерная революция, и потребность в программном обеспечении быстро превысила возможности программистов по его созданию. В академических кругах большие усилия были приложены к созданию более совершенного языка программирования. Однако, и это наиболее важно, все больше стало ощущаться влияние еще одного фактора. Компьютеры, наконец, получили достаточно широкое распространение, чтобы была достигнута "критическая масса". Компьютеры больше не находились за запертыми дверями. Впервые программисты получили буквально неограниченный доступ к своим компьютерам. Это дало свободу экспериментам. Программисты смогли также приступить к созданию своих собственных программных средств. Накануне создания языка C произошел качественный скачок в области компьютерных языков.

Изобретенный и впервые реализованный Деннисом Ритчи на компьютере DEC PDP-11, работающем под управлением операционной системы UNIX, язык C явился результатом процесса разработки, начавшегося с предшествующего языка

BCPL, разработанного Мартином Ричардсом. BCPL повлиял на язык, получивший название B, который был изобретен Кеном Томпсоном и в начале 70-х гг. привел к появлению языка C. В течение долгих лет фактическим стандартом языка C была его версия, которая поставлялась с операционной системой UNIX (описана в книге *Язык программирования C* Брайана Кернигана и Денниса Ритчи (2-е издание, И.Д. "Вилямс", 2007 г.)). Язык C был формально стандартизован в декабре 1989 г., когда Национальный институт стандартизации США (American National Standards Institute – ANSI) принял стандарт C.

Многие считают создание языка C началом современного этапа развития компьютерных языков. Он успешно объединил конфликтующие компоненты, которые доставляли столько неприятностей в предшествующих языках. Результатом явился мощный, эффективный, структурированный язык, изучение которого было сравнительно простым. Кроме того, ему была присуща еще одна, почти непостижимая особенность: он был языком программиста. До появления C языки программирования проектировались, в основном, либо в качестве академических упражнений, либо бюрократическими организациями. Язык C – иное дело. Он был спроектирован, реализован и разработан действительно работающими программистами и отражал их подход к программированию. Его функции были отлажены, проверены и многократно переработаны людьми, которые действительно использовали этот язык. В результате появился язык, который нравилось использовать программистам. Действительно C быстро приобрел много приверженцев, которые почти молились на него. Поэтому язык C получил быстрое и широкое признание в программистском сообществе. Короче говоря, C – это язык, разработанный программистами и для программистов. Как вы вскоре убедитесь, язык Java унаследовал эту особенность.

Следующий шаг: язык C++

В конце 70-х–начале 80-х гг. язык C стал превалирующим компьютерным языком программирования, и он продолжает широко применяться и в настоящее время. Поскольку C – удачный и удобный язык, может возникнуть вопрос, чем обусловлена потребность в чем-либо еще. Ответ: растущей *сложностью*. На протяжении всей истории развития программирования всевозрастающая сложность программ порождала потребность в более совершенных способах преодоления этой сложности. Язык C++ явился ответом на эту потребность. Чтобы лучше понять, почему потребность преодоления сложности программ является главной побудительной причиной создания языка C++, рассмотрим следующие факторы.

С момента изобретения компьютеров подходы к программированию коренным образом изменились. Например, когда компьютеры только появились, программирование осуществлялось изменением двоичных машинных инструкций вручную с панели управления компьютера. До тех пор, пока длина программ не превышала нескольких сотен инструкций, этот подход был вполне приемлем. С увеличением программ был изобретен язык ассемблера, который позволил программисту работать с большими, все более сложными программами, используя при этом символьные представления машинных инструкций. По мере того, как программы продолжали увеличиваться в объеме, появились языки высокого уровня, которые предоставили программисту дополнительные средства преодоления сложности программ.

Первым языком программирования, который получил широкое распространение, был, конечно же, FORTRAN. Хотя этот язык и явился первым впечатляющим шагом, его вряд ли можно считать языком, который способствует созданию четких и легких для понимания программ. 60-е гг. знаменовались рождением *структурного программирования*. Этот метод программирования наиболее ярко проявился в таких языках, как C. Использование структурированных языков впервые

предоставило программистам возможность достаточно легко создавать программы средней сложности. Однако даже при использовании методов структурного программирования по достижении проектом определенного размера его сложность начинала превышать ту, с которой программист мог справиться. К началу 80-х гг. сложность многих проектов начала превышать ту, с которой можно было справиться с использованием структурного подхода. Для решения этой проблемы был изобретен новый способ программирования, получивший название *объектно-ориентированного программирования* (ООП). Объектно-ориентированное программирование подробно рассматривается в последующих главах этой книги, но мы все же приведем краткое определение: ООП — это методология программирования, которая помогает организовывать сложные программы за счет использования наследования, инкапсуляции и полиморфизма.

Подведем итоги сказанному. Хотя C — один из основных мировых языков программирования, существует предел его способности справляться со сложностью программ. Как только размеры программы превышают определенное значение, она становится слишком сложной, чтобы ее можно было охватить как единое целое. Хотя точное значение этого предела зависит как от структуры самой программы, так и от подходов, используемых программистом, начиная с определенного момента любая программа становится слишком сложной для понимания и внесения изменений. Язык C++ предоставил возможности, которые позволили программисту преодолевать этот барьер, чтобы контролировать большие по размеру программы и управлять ими.

Язык C++ был изобретен Бьярне Страуструпом в 1979 г. во время его работы в компании Bell Laboratories в городе Мюррей-Хилл, шт. Нью-Джерси. Вначале Страуструп назвал новый язык "C with Classes" ("C с классами"). Однако в 1983 г. это название было изменено на C++. Язык C++ расширяет функциональные возможности языка C, добавляя в него объектно-ориентированные свойства. Поскольку язык C++ построен на основе C, он поддерживает все его возможности, атрибуты и преимущества. Это обстоятельство явилось главной причиной успешного распространения C++ в качестве языка программирования. Изобретение языка C++ не было попыткой создания совершенно нового языка программирования. Напротив, все усилия были направлены на усовершенствование уже существующего очень удачного языка.

Предпосылки создания языка Java

К концу 80-х–началу 90-х гг. объектно-ориентированное программирование с применением языка C++ стало основным методом программирования. Действительно, в течение некоторого непродолжительного времени казалось, что программисты, наконец, изобрели идеальный язык. Поскольку язык C++ сочетал в себе высокую эффективность и стилистические элементы языка C с объектно-ориентированным подходом, этот язык можно было использовать для создания самого широкого круга программ. Однако, как и в прошлом, уже вызревали факторы, которые должны были, в который раз, стимулировать развитие компьютерных языков. Пройдет еще несколько лет, и World Wide Web и Интернет достигнут критической массы. Это приведет к еще одной революции в программировании.

Создание языка Java

Начало разработке языка Java было положено в 1991 г. Джеймсом Гослингом (James Gosling), Патриком Нотоном (Patrick Naughton), Крисом Вартом (Chris

Warth), Эдом Франком (Ed Frank) и Майком Шериданом (Mike Sheridan), работавшими в компании Sun Microsystems, Inc. Разработка первой работающей версии заняла 18 месяцев. Вначале язык получил название "Oak" ("Дуб"), но в 1995 г. он был переименован в "Java". Между первой реализацией языка Oak в конце 1992 г. и публичным объявлением о создании Java весной 1995 г. множество других людей приняли участие в проектировании и развитии этого языка. Билл Джой (Bill Joy), Артур ван Хофф (Arthur van Hoff), Джонатан Пэйн (Jonathan Payne), Франк Йеллин (Frank Yellin) и Тим Линдхольм (Tim Lindholm) внесли основной вклад в развитие исходного прототипа.

Как ни странно, первоначальной побудительной причиной создания языка Java послужил вовсе не Интернет! Основная причина – потребность в независящем от платформы (т.е. архитектурно нейтральном) языке, который можно было бы использовать для создания программного обеспечения, встраиваемого в различные бытовые электронные устройства, такие как микроволновые печи и устройства дистанционного управления. Как ни трудно догадаться, в качестве контроллеров используется множество различных типов процессоров. Проблема применения языков C и C++ (как и большинства других языков) состоит в том, что написанные на них программы должны компилироваться для конкретной платформы. Хотя программы C++ могут быть скомпилированы практически для любого типа процессора, для этого требуется наличие полного компилятора C++, предназначенного для данного процессора. Проблема в том, что создание компиляторов обходится дорого и требует значительного времени. Поэтому требовалось более простое и экономически выгодное решение. Пытаясь найти такое решение, Гослинг и другие начали работу над переносимым, не зависящим от платформы языком, который можно было бы использовать для создания кода, пригодного для выполнения на различных процессорах в различных средах. Вскоре эти усилия привели к созданию языка Java.

Примерно в то же время, когда определялись основные характеристики языка Java, на сцену выступил второй, несомненно, более важный фактор, который должен был сыграть решающую роль в судьбе этого языка. Конечно же, этим вторым фактором была World Wide Web. Если бы формирование веб не происходило почти одновременно с реализацией Java, этот язык мог бы остаться полезным, но незамеченным языком программирования бытовых электронных устройств. Но с появлением World Wide Web язык Java вышел на передний рубеж проектирования компьютерных языков, поскольку веб также нуждался в переносимых программах.

Еще на заре своей карьеры большинство программистов твердо усвоили, что переносимые программы столь же недостижимы, сколь и желанны. В то время как потребность в средстве создания эффективных, переносимых (не зависящих от платформы) программ была почти столь же стара, как и сама отрасль программирования, она отодвигалась на задний план другими, более насущными, проблемами. Более того, поскольку большая часть самого мира компьютеров была разделена на три конкурирующих лагеря Intel, Microsoft и UNIX, большинство программистов оставались запертыми в своих аппаратно-программных "твердынях", что несколько снижало потребность в переносимом коде. Тем не менее с появлением Интернета и веб старая проблема переносимости снова возникла с еще большей актуальностью. В конце концов, Интернет представляет собой разнообразную и распределенную вселенную, заполненную множеством различных типов компьютеров, операционных систем и процессоров. Несмотря на то что к Интернету подключено множество типов платформ, пользователям желательно, чтобы все они могли выполнять одинаковые программы. То, что в начале было неприятной, но не слишком насущной проблемой, превратилось в потребность первостепенной важности.

К 1993 г. членам группы проектирования Java стало очевидно, что проблемы переносимости, часто возникающие при создании кода, предназначенного для встраивания в контроллеры, возникают также и при попытках создания кода для Интернета. Фактически та же проблема, для решения которой в малом масштабе предназначался язык Java, в большем масштабе была актуальна и в среде Интернета. Понимание этого обстоятельства вынудило разработчиков языка Java перенести свое внимание с бытовой электроники на программирование для Интернета. Таким образом, хотя потребность в архитектурно нейтральном языке программирования послужила своего рода "начальной искрой", Интернет обеспечил крупномасштабный успех Java.

Как уже упоминалось, язык Java наследует многие из своих характеристик от языков C и C++. Это сделано намеренно. Разработчики Java знали, что использование знакомого синтаксиса C и повторение объектно-ориентированных свойств C++ должно было сделать их язык привлекательным для миллионов опытных программистов на C/C++. Помимо внешнего сходства, язык Java использует ряд других атрибутов, которые способствовали успеху языков C и C++. Во-первых, язык Java был спроектирован, проверен и усовершенствован настоящими работающими программистами. Этот язык построен с учетом потребностей и опыта людей, которые его создали. Таким образом, Java — это язык программистов. Во-вторых, Java целостен и логически непротиворечив. В-третьих, если не учитывать ограничения, накладываемые средой Интернета, Java предоставляет программисту полный контроль над программой. Если программирование выполняется правильно, это непосредственно отражается в программах. В равной степени справедливо и обратное. Иначе говоря, Java не является языком тренажера. Это язык профессиональных программистов.

Из-за сходства характеристик языков Java и C, кое-кто склонен считать Java просто "версией языка C++ для Интернета". Однако это серьезное заблуждение. Языку Java присущи значительные практические и концептуальные отличия. Хотя и верно, что язык C++ оказал влияние на характеристики языка Java, последний не является усовершенствованной версией C++. Например, Java не обладает совместимостью с C++. Конечно, сходство с языком C++ значительно, и в программе Java программист C++ будет чувствовать себя как дома. Вместе с тем, Java не предназначен служить заменой C++. Язык Java предназначен для решения одного набора проблем, а C++ — для решения другого. Еще длительное время оба эти языка неизбежно будут сосуществовать.

Как было отмечено в начале этой главы, развитие компьютерных языков обусловлено двумя причинами: необходимостью адаптации к изменениям в среде и необходимостью реализации новых идей в области программирования. Изменением среды, которое обусловило потребность в языке, подобном Java, была потребность в независящих от платформы программах, предназначенных для распространения в Интернете. Однако Java изменяет также подход к написанию программ. В частности, Java углубил и усовершенствовал объектно-ориентированный подход, использованный в C++, добавил в него поддержку многопоточной обработки и предоставил библиотеку, которая упростила доступ к Интернету. Однако столь поразительный успех Java обусловлен не теми или иными его отдельными особенностями, а их совокупностью как языка в целом. Он явился прекрасным ответом на потребности в то время лишь зарождающейся среды в высшей степени распределенных компьютерных систем. В области разработки программ для Интернета язык Java стал тем, чем язык C был для системного программирования: революционной силой, которая изменила мир.

Связь с языком C#

Многообразие и большие возможности языка Java продолжают оказывать влияние на всю разработку компьютерных языков. Многие из его новаторских характеристик, конструкций и концепций становятся неотъемлемой частью фундамента любого нового языка. Просто успех Java слишком значителен, чтобы его можно было игнорировать.

Вероятно, наиболее наглядным примером влияния языка Java на программирование служит язык C#. Созданный в компании Microsoft для поддержки инфраструктуры .NET Framework, язык C# тесно связан с Java. Например, оба эти языка используют одинаковый общий синтаксис, поддерживают распределенное программирование и работают с одной и той же объектной моделью. Конечно, между Java и C# существует ряд различий, но в целом эти языки "выглядят" очень похожими. На сегодняшний день это "перекрестное опыление" между Java и C# — наилучшее доказательство того, что язык Java коренным образом изменил представление о компьютерных языках и их применении.

Как язык Java изменил Интернет

Интернет способствовал выдвижению языка Java на передовые рубежи программирования, а язык Java, в свою очередь, оказал сильнейшее влияние на Интернет. Кроме того, что язык Java упростил создание программ для Интернета в целом, он привел к появлению нового типа предназначенных для работы в сетях программ, получивших название аплетов, которые изменили понятие содержимого для сетевой среды. Кроме того, язык Java позволил решить две наиболее острые проблемы программирования, связанные с Интернетом, — переносимость и безопасность. Рассмотрим каждую из этих проблем.

Аплеты Java

Аплет — это особый вид программы Java, предназначенный для передачи по Интернету и автоматического выполнения совместимым с Java веб-браузером. Более того, аплет загружается по требованию, без необходимости дальнейшего взаимодействия с пользователем. Если пользователь щелкает на ссылке, которая содержит аплет, он автоматически загружается и запускается в браузере. Аплеты создаются в виде небольших программ. Как правило, они используются для отображения данных, предоставляемых сервером, обработки действий пользователя или выполнения простых функций, таких как вычисление процентов по кредитам, которые выполняются локально, а не на сервере. По сути, аплет позволяет перенести ряд функций с сервера на клиент.

Появление аплетов изменило программирование приложений Интернета, поскольку они расширили совокупность объектов, которые можно свободно перемещать по киберпространству. Если говорить в целом, между сервером и клиентом передаются две большие категории объектов: пассивная информация и динамические, активные, программы. Например, чтение сообщений электронной почты подразумевает просмотр пассивных данных. Даже при загрузке программы ее код — пассивные данные, которые остаются таковыми, вплоть до момента выполнения. И напротив, аплет представляет собой динамическую, автоматически выполняющуюся программу. Такая программа является активным агентом на клиентском компьютере, хотя она и инициируется сервером.

Насколько желательно, чтобы программы были динамическими, как это имеет место при использовании сетевых программ, настолько же они представляют серьезные проблемы с точки зрения безопасности и переносимости. Очевидно, что компьютер клиента необходимо обезопасить от нанесения ему ущерба программой, которая загружается в него, а затем автоматически запускается. Кроме того, такая программа должна быть способна выполняться в различных аппаратных средах и под управлением различных операционных систем. Как читатели вскоре убедятся, язык Java решает эти проблемы эффективно и элегантно. Рассмотрим их подробнее.

Безопасность

Как, вероятно, известно читателям, каждая загрузка "обычной" программы сопряжена с риском, поскольку загружаемый код может содержать вирус, "троянского коня" или вредоносный код. Корень проблемы в том, что вредоносный код может выполнить свое "черное" дело, поскольку получает несанкционированный доступ к системным ресурсам. Например, просматривая содержимое локальной файловой системы компьютера, программа вируса может собирать конфиденциальную информацию, такую как номера кредитных карточек, сведения о состоянии банковских счетов и пароли. Для обеспечения безопасности загрузки и выполнения аплетов Java на компьютере клиента было необходимо воспрепятствовать аплетам предпринимать подобные атаки.

Java обеспечивает эту защиту, заключая аплет в среду исполнения Java и не предоставляя ему доступ к другим частям операционной системы компьютера. (Способы достижения этого рассматриваются в последующих разделах.) Возможность загрузки аплетов с сохранением при этом уверенности в невозможности нанесения вреда системе и нарушения системы безопасности многие эксперты и пользователи считают наиболее новаторским аспектом Java.

Переносимость

Переносимость — основная особенность Интернета, поскольку эта глобальная сеть соединяет множество различных типов компьютеров и операционных систем. Чтобы программа Java могла выполняться буквально на любом компьютере, подключенном к Интернету, требовался метод обеспечения выполнения этой программы в различных системах. Например, применительно к аплету это означает, что один и тот же аплет должен иметь возможность загружаться и выполняться на широком множестве процессоров, операционных систем и браузеров, подключенных к Интернету. Создание различных версий аплетов для различных компьютеров совершенно нерационально. *Один и тот же* код должен работать на *всех* компьютерах. Поэтому требовался какой-то механизм для создания переносимого выполняемого кода. Как вы вскоре убедитесь, тот же механизм, который способствует обеспечению безопасности, способствует также созданию переносимых программ.

Магия Java: код виртуальной машины

Основная особенность, которая позволяет языку Java решать описанные проблемы обеспечения безопасности и переносимости программ, состоит в том, что вывод компилятора Java не является исполняемым кодом. Скорее, он представляет собой так называемый код виртуальной машины. *Код виртуальной машины* (by-

tecode) — это в высшей степени оптимизированный набор инструкций, предназначенных для исполнения системой времени выполнения Java, называемой *виртуальной машиной Java* (Java Virtual Machine — JVM). Собственно говоря, первоначальная версия JVM разрабатывалась в качестве *интерпретатора кода виртуальной машины*. Это может вызывать определенное удивление, поскольку по соображениям обеспечения максимальной производительности многие современные языки призваны создавать исполняемый код.

Однако то, что программа Java интерпретируется машиной JVM, помогает решать основные проблемы, связанные с программами, предназначенными для веб. И вот почему.

Трансляция программы Java в код виртуальной машины значительно упрощает ее выполнение в широком множестве сред, поскольку на каждой платформе необходимо реализовать только JVM. Как только в данной системе появляется пакет времени выполнения, в ней можно исполнять любую программу Java. Следует помнить, что хотя на разных платформах особенности реализации машины JVM могут быть различными, все они могут выполнять обработку одного и того же кода виртуальной машины. Если бы программа Java компилировалась в машинозависимый код, для каждого типа процессоров, подключенных к Интернету, должны были бы существовать отдельные версии одной и той же программы. Понятно, что такое решение неприемлемо. Таким образом, выполнение кода виртуальной машины машиной JVM — простейший способ создания действительно переносимых программ.

То, что программа Java выполняется машиной JVM, способствует также повышению ее безопасности. Поскольку машина JVM управляет выполнением программы, она может изолировать программу и воспрепятствовать порождению ею побочных эффектов вне данной системы. Как вы убедитесь, ряд ограничений, существующих в языке Java, также способствует повышению безопасности.

В общем случае, когда программа компилируется в промежуточную форму, а затем интерпретируется виртуальной машиной, она выполняется медленнее, чем если бы она была скомпилирована в исполняемый код. Однако при использовании языка Java различие в производительности не слишком велико. Поскольку код виртуальной машины в высокой степени оптимизирован, его применение позволяет машине JVM выполнять программы значительно быстрее, чем можно было ожидать.

Хотя язык Java был задуман в качестве интерпретируемого языка, ничто не препятствует ему выполнять компиляцию кода виртуальной машины в машинозависимый код "на лету" для повышения производительности. Поэтому вскоре после выпуска Java появилась технология HotSpot. Эта технология предоставляет оперативный компилятор (JustIn-Time — JIT) кода виртуальной машины. Когда JIT-компилятор является составной частью машины JVM, избранные фрагменты кода виртуальной машины один за другим компилируются в исполняемый код в реальном времени, по соответствующим запросам. Важно понимать, что одновременная компиляция всей программы Java в исполняемый код нецелесообразна, поскольку Java выполняет различные проверки, которые могут быть осуществлены только во время выполнения. Вместо этого во время работы JIT-компилятор компилирует код по мере необходимости. Более того, компилируются не все фрагменты кода виртуальной машины, а только те, которым компиляция принесет выгоду. Остальной код просто интерпретируется. Однако подход JIT-компиляции все же обеспечивает значительное повышение производительности. Даже в случае применения к коду виртуальной машины динамической компиляции, характеристики переносимости и безопасности сохраняются, поскольку машина JVM по-прежнему отвечает за целостность среды исполнения.

Сервлеты: серверные программы Java

Как ни полезны аплеты, они — всего лишь половина системы "клиент/сервер". Вскоре после появления языка Java стало очевидно, что он может пригодиться и на серверах. В результате появились *сервлеты* (servlet). Сервлет — это небольшая программа, выполняемая на сервере. Подобно тому как аплеты динамически расширяют функциональные возможности веб-браузера, сервлеты динамически расширяют функциональные возможности веб-сервера. Таким образом, с появлением сервлетов язык Java распространился на оба конца соединения "клиент/сервер".

Сервлеты служат для создания динамически генерируемого содержимого, которое затем обслуживает клиента. Например, интерактивный склад может использовать сервлет для поиска стоимости товара в базе данных. Затем информация о цене используется для динамического создания веб-страницы, отправляемой браузеру. Хотя динамически создаваемое содержимое доступно также при помощи таких механизмов, как CGI (Common Gateway Interface — общий шлюзовой интерфейс), сервлет обеспечивает ряд преимуществ, в том числе — повышение производительности.

Поскольку сервлеты (подобно всем программам Java) компилируются в код виртуальной машины и выполняются машиной JVM, они в высшей степени переносимы. Следовательно, один и тот же сервлет может применяться в различных серверных средах. Единственные необходимые условия для этого — поддержка сервером машины JVM и контейнера сервлета.

Терминология, связанная с Java

Рассмотрение истории создания и развития языка Java было бы неполным без описания специфичной терминологии Java. Основные факторы, обусловившие изобретение Java, — необходимость обеспечения переносимости и безопасности, однако другие факторы также сыграли свою роль в формировании окончательной версии языка. Группа разработки Java обобщила основные понятия в следующем перечне терминов:

- простота;
- безопасность;
- переносимость;
- объектная ориентированность;
- устойчивость;
- многопоточность;
- архитектурная нейтральность;
- интерпретируемость;
- высокая производительность;
- распределенный характер;
- динамический характер.

Мы уже рассмотрели такие термины, как безопасность и переносимость. А теперь представим значения остальных терминов.

Простота

Язык Java был задуман в качестве простого в изучении и эффективного в использовании профессиональными программистами языка. Для тех, кто обладает определенным опытом программирования, овладение языком Java не представит особой сложности. Если же вы уже знакомы с базовыми концепциями объектно-ориентированного программирования, изучение Java будет еще проще. А для опытного программиста на C++ переход к Java вообще потребует минимум усилий. Поскольку Java наследует синтаксис C/C++ и многие объектно-ориентированные свойства C++, для большинства программистов изучение Java не представит сложности.

Объектная ориентированность

Хотя предшественники языка Java и оказали влияние на его архитектуру и синтаксис, при его проектировании не ставилась задача совместимости по исходному коду с каким-либо другим языком. Это позволило группе разработки Java выполнять проектирование, что называется, с чистого листа. Одним из следствий этого явился четкий, практичный, прагматичный подход к объектам. Притом что Java позаимствовал свойства многих удачных объектно-программных сред, разработанных на протяжении нескольких последних десятилетий, в нем удалось достичь баланса между строгим соблюдением концепции "все компоненты программы — объекты" и более прагматичной моделью "прочь с дороги". Объектная модель Java проста и легко расширяема. В то же время элементарные типы, такие как целые числа, сохраняются в виде высокопроизводительных компонентов, не являющихся объектами.

Устойчивость

Многоплатформенная среда веб предъявляет к программам повышенные требования, поскольку они должны надежно выполняться в разнообразных системах. Поэтому способность создавать устойчивые программы была одним из главных приоритетов при проектировании Java. Для обеспечения надежности Java накладывает ряд ограничений в нескольких наиболее важных областях, что вынуждает программиста выявлять ошибки на ранних этапах разработки программы. В то же время Java избавляет от беспокойства по поводу многих наиболее часто встречающихся ошибок программирования. Поскольку Java — строго типизированный язык, проверка кода выполняется во время компиляции. Однако проверка кода осуществляется и во время выполнения. В результате многие трудно обнаруживаемые программные ошибки, которые часто ведут к возникновению трудно воспроизводимых ситуаций времени выполнения, в программе Java попросту невозможны. Предсказуемость кода в различных ситуациях — одна из основных особенностей Java.

Чтобы понять, как достигается устойчивость программ Java, рассмотрим две основные причины программных сбоев: ошибки управления памятью и неправильная обработка исключений (т.е. ошибки времени выполнения). В традиционных средах создания программ управление памятью — сложная и трудоемкая задача. Например, в среде C/C++ программист должен вручную резервировать и освобождать всю динамически распределяемую память. Иногда это ведет к возникновению проблем, поскольку программисты либо забывают освободить ранее зарезервированную память, либо, что еще хуже, пытаются освободить участок памяти, все еще используемый другой частью кода. Java полностью исключает такие ситуации, автоматически управляя резервированием и освобождением памяти. (Фактически освобождение выполняется полностью автоматически, поскольку

Java предоставляет функцию сбора "мусора" в отношении неиспользуемых объектов.) В традиционных средах условия исключений часто возникают в таких ситуациях, как деление на нуль или "файл не найден", и управление ими должно осуществляться с помощью громоздких и трудных для понимания конструкций. Java облегчает выполнение этой задачи, предлагая объектно-ориентированный механизм обработки исключений. В хорошо написанной программе Java все ошибки времени выполнения могут — и должны — управляться самой программой.

Многопоточность

Язык Java был разработан в ответ на потребность создания интерактивных сетевых программ. Для достижения этой цели Java поддерживает написание многопоточных программ, которые могут одновременно выполнять много действий. Система времени выполнения Java содержит изящное, но вместе с тем сложное решение задачи синхронизации множества процессов, которое позволяет создавать действующие без перебоев интерактивные системы. Простой в применении подход к организации многопоточной обработки, реализованный в Java, позволяет программисту сосредоточивать свое внимание на конкретном поведении программы, а не на создании многозадачной подсистемы.

Архитектурная нейтральность

Основной задачей, которую ставили перед собой разработчики Java, было обеспечение долговечности и переносимости кода. Одной из главных проблем, стоявших перед программистами во время создания языка Java, было отсутствие гарантий того, что код, созданный сегодня, будет успешно выполняться завтра, даже на том же самом компьютере. Операционные системы и процессоры модернизируются, а все изменения в основных системных ресурсах могут приводить к неработоспособности программ. Пытаясь изменить эту ситуацию, проектировщики приняли ряд жестких решений в языке Java и виртуальной машине Java. Они поставили перед собой цель, чтобы "программы создавались лишь однажды, в любой среде, в любое время и навсегда". В значительной степени эта цель была достигнута.

Интерпретируемость и высокая производительность

Как уже говорилось, выполняя компиляцию программ в промежуточное представление, называемое кодом виртуальной машины, Java позволяет создавать многоплатформенные программы. Этот код может выполняться в любой системе, которая реализует виртуальную машину Java. Самые первые попытки получения многоплатформенных решений привели к достижению поставленной цели за счет снижения производительности. Как пояснялось ранее, код виртуальной машины Java был тщательно спроектирован так, чтобы за счет использования JIT-компиляции его можно было с высокой производительностью легко преобразовывать в машинозависимый код. Системы времени выполнения Java, которые предоставляют эту функцию, сохраняют все преимущества кода, не зависящего от платформы.

Распределенный характер

Язык Java предназначен для распределенной среды Интернет, поскольку он поддерживает протоколы семейства TCP/IP. Фактически обращение к ресурсу че-

рез адрес URL не очень отличается от обращения к файлу. Java поддерживает также *дистанционный вызов методов* (Remote Method Invocation – RMI). Это свойство позволяет программам вызывать методы по сети.

Динамический характер

Программы Java содержат значительный объем информации времени выполнения, которая используется для проверки полномочий и предоставления доступа к объектам во время выполнения. Это позволяет выполнять безопасное и технически оправданное динамическое связывание кода. Это обстоятельство исключительно важно для устойчивости среды Java, в которой небольшие фрагменты кода виртуальной машины могут динамически обновляться в действующей системе.

Эволюция языка Java

Первоначальная версия Java не содержала никаких особо революционных решений, но она не ознаменовала собой завершение эры быстрого совершенствования этого языка.

В отличие от большинства других систем программирования, совершенствование которых происходило небольшими, последовательными шагами, язык Java продолжает стремительно развиваться. Уже вскоре после выпуска версии Java 1.0 разработчики создали версию Java 1.1. Добавленные в эту версию функциональные возможности значительно превосходили те, которые можно было ожидать, судя по изменению младшего номера версии. Разработчики добавили много новых библиотечных элементов, переопределили способ обработки событий и изменили конфигурацию многих свойств библиотеки версии 1.0. Кроме того, они отказались от нескольких свойств (признанных устаревшими), которые первоначально были определены в Java 1.0. Таким образом, в версии Java 1.1 были как добавлены новые атрибуты, так и удалены некоторые, определенные в первоначальной спецификации.

Следующей базовой версией Java стала версия Java 2, где "2" означает "второе поколение". Создание Java 2 явилось знаменательным событием, означавшим начало "современной эры" Java. Первой версии Java 2 был присвоен номер 1.2. Это может казаться несколько странным. Дело в том, что вначале номер относился к внутреннему номеру версии библиотек Java, но затем он был распространен на всю версию в целом. С появлением версии Java 2 компания Sun начала выпускать программное обеспечение Java в виде пакета J2SE (Java 2 Platform Standard Edition – Стандартная версия платформы Java 2), и теперь номера версий применяются к этому продукту.

В Java 2 была добавлена поддержка ряда новых средств, таких как Swing и Collections Framework. Кроме того, были усовершенствованы виртуальная машина Java и различные средства программирования. Из Java 2 был исключен также ряд свойств. Наибольшие изменения претерпел класс потока `Thread`, в котором методы `suspend()`, `resume()` и `stop()` были представлены как устаревшие.

Версия J2SE 1.3 была первой серьезной модернизацией первоначальной версии Java J2SE. В основном, модернизация заключалась в расширении существующих функциональных возможностей и "уплотнении" среды 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 три предшествующие базовые версии, это не делало ее менее важной.

Версия J2SE 1.4 продолжила совершенствование языка Java. Эта версия содержала несколько важных модернизаций, усовершенствований и добавлений.

Например, в нее было добавлено новое ключевое слово `assert`, цепочки исключений и подсистема ввода-вывода на основе каналов. Изменения были внесены и в инфраструктуру Collections Framework, и сетевые классы. Эта версия содержала также множество небольших изменений. Несмотря на значительное количество новых функциональных возможностей, версия 1.4 сохранила почти стопроцентную совместимость по исходному коду с предшествующими версиями.

В следующей версии Java, именуемой J2SE 5, был внесен ряд революционных изменений. В отличие от большинства предшествующих модернизаций Java, которые предоставляли важные, но постепенные усовершенствования, J2SE 5 коренным образом расширяет область применения, возможности и диапазон языка. Чтобы оценить объем изменений, внесенных в язык Java в версии J2SE 5, ознакомьтесь с перечнем основных новых функциональных возможностей:

- обобщения;
- аннотации;
- автоупаковка и автораспаковка;
- перечисления;
- усовершенствованный, поддерживающий стиль `for-each`, цикл `for`;
- список аргументов переменной длины (varargs);
- статический импорт;
- форматированный ввод-вывод;
- утилиты параллельной обработки.

В этом перечне не указаны незначительные изменения или постепенные усовершенствования. Каждый пункт перечня представлял значительное добавление в языке Java. Одни из них, такие как обобщения, усовершенствованный цикл `for` и список аргументов переменной длины, представляли новые синтаксические элементы. Другие, такие как автоупаковка и автораспаковка, изменяли семантику языка. Аннотации внесли в программирование совершенно новое измерение. В любом случае влияние всех этих добавлений вышло за рамки их прямого эффекта. Они полностью изменили сам характер языка Java.

Важность новых функциональных возможностей нашла отражение в примененном номере версии — "5". Если следовать привычной логике, следующим номером версии Java должен был быть 1.5. Однако новые свойства столь значительны, что переход от версии 1.4 к версии 1.5 не отражал бы масштаб внесенных изменений. Поэтому, чтобы подчеркнуть значимость этого события, в компании Sun решили присвоить новой версии номер 5. Поэтому версия продукта была названа J2SE 5, а комплект разработчика — JDK 5. Тем не менее для сохранения единообразия в компании Sun решили использовать номер 1.5 в качестве внутреннего номера версии, называемого также номером *версии разработки*. Цифру 5 в обозначении версии называют номером *версии продукта*.

Следующая версия Java получила название SE 6. С выходом этой версии компания Sun решила в очередной раз изменить название платформы Java. В названии была опущена цифра 2. Таким образом, теперь платформа называется *Java SE*, а официальное название продукта — *Java Platform, Standard Edition 6* (Платформа Java, стандартная версия 6). Комплект разработчика Java был назван JDK 6. Как и в обозначении версии J2SE 5, цифра 6 в названии Java SE 6 означает номер версии продукта. Внутренним номером разработки этой версии является 1.6.

Версия Java SE 6 была построена на основе версии J2SE 5 с рядом дальнейших усовершенствований. Она не содержала дополнений к числу основных функций языка Java, но расширяла библиотеки API, добавляя несколько новых пакетов и предоставляя ряд усовершенствований времени выполнения. Было сделано еще несколько модификаций и внесено несколько дополнений. В целом версия Java SE 6 призвана закрепить достижения, полученные в J2SE 5.

Java SE 7

Новейший выпуск Java, Java SE 7, с комплектом разработчика Java JDK 7 и внутренним номером версии 1.7 — первый главный выпуск Java от Sun Microsystems, который был приобретен Oracle (процесс начался в апреле 2009 года и завершился в январе 2010 года). Java SE 7 содержит много новых средств, включая существенные дополнения языка и библиотек API. Усовершенствована система исполнения программ Java, включена также поддержка языков, отличных от Java, но программистам Java больше интересны дополнения в области языка и библиотек.

Проект Project Coin должен объединить множество небольших изменений языка Java, которые будут включены в JDK 7. Хотя все эти новые средства вместе называют "небольшими", эффект этих изменений весьма велик с точки зрения кода, на который они воздействуют. Фактически для большинства программистов эти изменения могут стать самой важной новой возможностью Java SE 7. Вот список новых средств языка.

- Тип `String` теперь может контролировать оператор `switch`.
- Бинарные целочисленные литералы.
- Символы подчеркивания в числовых литералах.
- Расширенный оператор `try`, называемый `try-с-ресурсами`, обеспечивающий автоматическое управление ресурсами. (Например, потоки теперь могут быть закрыты автоматически, когда они больше не нужны.)
- Выведение типов (при помощи оператора `<>`) при создании экземпляра обобщения.
- Улучшенная обработка исключений, при которой два или больше исключений могут быть обработаны одним блоком `catch` (мультиобработчиком), и лучший контроль соответствия типов повторно передаваемых исключений.
- Хотя синтаксис не изменился, предупреждения компилятора, связанные с некоторыми типами методов и переменным количеством аргументов, были улучшены. Кроме того, вы получаете больше контроля над предупреждениями.

Как можно заметить, даже притом что средства Project Coin считаются небольшими изменениями языка, их преимуществ, в целом, намного больше, чем подразумевает слово "небольшой". В частности оператор `try-с-ресурсами` существенно влияет на способ создания кода, ориентированного на потоки. Кроме того, способность использовать строки в операторе `switch` является долгожданным усовершенствованием, которое упростит код во многих ситуациях.

В Java SE 7 внесено несколько дополнений в библиотеку API Java. Важнейшими из них являются усовершенствования NIO Framework и дополнения для Fork/Join Framework. NIO (первоначально именуемый как *New I/O*) был добавлен к Java в версии 1.4. Однако изменения, предложенные в Java SE 7, существенно увеличили его возможности. Причем настолько существенно, что зачастую используется термин *NIO.2*.

Инфраструктура Fork/Join Framework оказывает существенную поддержку для *параллельного программирования* (parallel programming). Параллельное программирование – это термин, используемый обычно для описания технологий, повышающих эффективность использования компьютеров с несколькими процессорами, включая многоядерные системы. Преимущества, предлагаемые многоядерными системами, способны в перспективе существенно увеличить производительность программ. Инфраструктура Fork/Join Framework обеспечивает параллельное программирование в таких областях, как

- упрощение создания и использования задач, способных выполняться одновременно;
- автоматическое использование нескольких процессоров.

Следовательно, используя инфраструктуру Fork/Join Framework, вы можете создавать приложения, которые автоматически используют процессоры, доступные в среде исполнения. Конечно, не все алгоритмы являются параллельными, но для остальных может быть получено существенное преимущество в скорости выполнения.

Материал этой книги был изменен так, чтобы отразить возможности Java SE 7 со всеми новыми средствами, модификациями и дополнениями, обозначенными повсюду.

Культура инновации

С самого начала язык Java оказался в центре культуры инновации. Его первоначальная версия изменила подход к программированию для Интернета. Виртуальная машина Java (JVM) и код виртуальной машины изменили представление о безопасности и переносимости. Аплет (а затем и сервлет) вдохнули жизнь в веб. Процесс Java Community Process (JCP) изменил способ ассимиляции новых идей в языке. Поскольку язык Java используется для программного обеспечения Android, он стал частью революции смартфонов. Никогда мир языка Java не оставался неизменным в течение длительного времени. На момент подготовки этого издания версия Java SE 7 являлась самой новой в непрекращающемся динамичном развитии языка Java.

ГЛАВА 2

Обзор языка Java

Как и во всех компьютерных языках, элементы Java существуют не сами по себе. Скорее, они работают совместно, образуя язык в целом. Однако эта взаимосвязанность может затруднять описание какого-то одного аспекта Java без рассмотрения при этом нескольких других. Зачастую для понимания одного свойства необходимы знания о другом. Поэтому в настоящей главе представлен краткий обзор нескольких основных свойств языка Java. Приведенный здесь материал послужит отправной точкой, которая позволит создавать и понимать простые программы. Большинство рассмотренных в этой главе тем будет подробнее рассмотрено в остальных главах этой части.

Объектно-ориентированное программирование

Основной характеристикой языка Java является объектно-ориентированное программирование (ООП). Фактически все программы Java являются в какой-то мере объектно-ориентированными. Язык Java связан с ООП настолько тесно, что прежде чем приступать к написанию на нем даже простейших программ, следует вначале ознакомиться с базовыми принципами ООП. Поэтому начнем с рассмотрения теоретических аспектов ООП.

Две концепции

Все компьютерные программы состоят из двух элементов: кода и данных. Более того, концептуально программа может быть организована вокруг своего кода или своих данных. Другими словами, организация одних программ определяется тем "что происходит", а других – тем, "на что оказывается влияние". Существует две концепции создания программы. Первую называют *моделью, ориентированной на процессы*. Этот подход характеризует программу в виде последовательностей линейных шагов (т.е. кода). Модель, ориентированную на процессы, можно рассматривать в качестве *кода, воздействующего на данные*. Процедурные языки вроде C достаточно успешно используют эту модель. Однако, как было отмечено в главе 1, этот подход порождает ряд проблем с увеличением размеров и сложности программ.

Для преодоления увеличения сложности была начата разработка подхода, названного *объектно-ориентированным программированием*. Объектно-ориентированное программирование организует программу вокруг ее данных (т.е. объектов) и набора подробно определенных интерфейсов к этим данным. Объектно-ориентированную программу можно характеризовать *как данные, управляющие доступом к коду*. Как будет показано в дальнейшем, передавая функции управления данными, можно получить несколько организационных преимуществ.

Абстракция

Важный элемент объектно-ориентированного программирования — *абстракция*. Человеку свойственно представлять сложные явления и объекты при помощи абстракции. Например, люди представляют себе автомобиль не в виде набора десятков тысяч отдельных деталей, а в виде совершенно определенного объекта, имеющего собственное уникальное поведение. Эта абстракция позволяет не задумываться о сложности деталей, образующих автомобиль, скажем, при поездке в магазин. Мы можем не обращать внимания на подробности работы двигателя, коробки передач и тормозной системы. Вместо этого объект можно использовать как единое целое.

Мощное средство применения абстракции — иерархические классификации. Это позволяет упрощать семантику сложных систем, разбивая их на более пригодные для управления фрагменты. Внешне автомобиль выглядит единым объектом. Но стоит заглянуть внутрь, как становится ясно, что он состоит из нескольких подсистем: рулевого управления, тормозов, аудиосистемы, привязных ремней, обогревателя, навигатора и т.п. Каждая из этих подсистем, в свою очередь, собрана из более специализированных узлов. Например, аудиосистема состоит из радиоприемника, проигрывателя компакт-дисков и/или устройства воспроизведения аудиокассет. Суть сказанного в том, что сложную структуру автомобиля (или любой другой сложной системы) можно описать с помощью иерархических абстракций.

Иерархические абстракции сложных систем можно применять и к компьютерным программам. За счет абстракции данные традиционной, ориентированной на процессы, программы можно преобразовать в составляющие ее объекты. При этом последовательность этапов процесса может быть преобразована в коллекцию сообщений, передаваемых между этими объектами. Таким образом, каждый из этих объектов описывает свое уникальное поведение. Эти объекты можно считать конкретными элементами, которые отвечают на сообщения, указывающие им о необходимости *выполнить что-либо*. Сказанное — суть объектно-ориентированного программирования.

Концепции объектной ориентированности лежат как в основе языка Java, так и восприятия мира человеком. Важно понимать, как эти концепции реализуются в программах. Как вы увидите, объектно-ориентированное программирование — мощная и естественная концепция создания программ, которые способны пережить неизбежные изменения, сопровождающие жизненный цикл любого крупного программного проекта, включая создание концепции, рост и старение. Например, при наличии тщательно определенных объектов и четких, надежных интерфейсов к этим объектам можно безбоязненно и без особых проблем извлекать или заменять части старой системы.

Три принципа ООП

Все языки объектно-ориентированного программирования предоставляют механизмы, которые облегчают реализацию объектно-ориентированной модели. Этими механизмами являются инкапсуляция, наследование и полиморфизм. Рассмотрим эти концепции.

Инкапсуляция

Механизм, связывающий код и данные, которыми он манипулирует, защищая оба эти компонента от внешнего вмешательства и злоупотреблений, является *ин-*

капсуляцией. Инкапсуляцию можно считать защитной оболочкой, которая предохраняет код и данные от произвольного доступа со стороны другого кода, находящегося снаружи оболочки. Доступ к коду и данным, находящимся внутри оболочки, строго контролируется тщательно определенным интерфейсом. Чтобы провести аналогию с реальным миром, рассмотрим автоматическую коробку передач автомобиля. Она инкапсулирует сотни бит информации об автомобиле, такой как степень ускорения, крутизна поверхности, по которой совершается движение, и положение рычага переключения скоростей. Пользователь (водитель) может влиять на эту сложную инкапсуляцию только одним методом: перемещая рычаг переключения скоростей. На коробку передач нельзя влиять, например, при помощи индикатора поворота или дворников. Таким образом, рычаг переключения скоростей — строго определенный (а в действительности единственный) интерфейс к коробке передач. Более того, происходящее внутри коробки передач не влияет на объекты, находящиеся вне ее. Например, переключение передач не включает фары! Поскольку функция автоматического переключения передач инкапсулирована, десятки изготовителей автомобилей могут реализовать ее как угодно. Однако с точки зрения водителя все эти коробки передач работают одинаково. Аналогичную идею можно применять к программированию. Сила инкапсулированного кода в том, что все знают, как к нему можно получить доступ, а следовательно, могут его использовать независимо от нюансов реализации и не опасаясь неожиданных побочных эффектов.

В языке Java основой инкапсуляции является класс. Хотя подробнее мы рассмотрим классы в последующих главах книги, сейчас полезно ознакомиться со следующим кратким описанием. *Класс* определяет структуру и поведение (данные и код), которые будут совместно использоваться набором объектов. Каждый объект данного класса содержит структуру и поведение, которые определены классом, как если бы объект был "отлит" в форме класса. Поэтому иногда объекты называют *экземплярами класса*. Таким образом, класс — это логическая конструкция, а объект имеет физическое воплощение.

При создании класса определяют код и данные, которые образуют этот класс. Совокупность этих элементов называют *членами* класса. В частности, определенные классом данные называют *переменными-членами* или *переменными экземпляра*. Код, который выполняет действия по отношению к данным, называют *методами-членами* или просто *методами*. (То, что программисты на Java называют *методом*, программисты на C/C++ называют *функциями*.) В правильно написанных программах Java методы определяют способы использования переменных-членов. Это означает, что поведение и интерфейс класса определяются методами, которые выполняют действия по отношению к данным его экземпляра.

Поскольку назначение класса — инкапсуляция сложной структуры программы, существуют механизмы сокрытия сложной структуры реализации внутри класса. Каждый метод или переменная в классе могут быть помечены как закрытые или открытые. *Открытый* интерфейс класса представляет все, что должны или могут знать внешние пользователи класса. *Закрытые* методы и данные могут быть доступны только для кода, который является членом данного класса. Следовательно, любой другой код, не являющийся членом класса, не может получать доступ к закрытому методу или переменной. Поскольку закрытые члены класса доступны другим частям программы только через открытые методы класса, можно быть уверенным в невозможности выполнения неправомерных действий. Конечно, это означает, что открытый интерфейс должен быть тщательно спроектирован и не должен раскрывать лишних нюансов внутренней работы класса (рис 2.1).

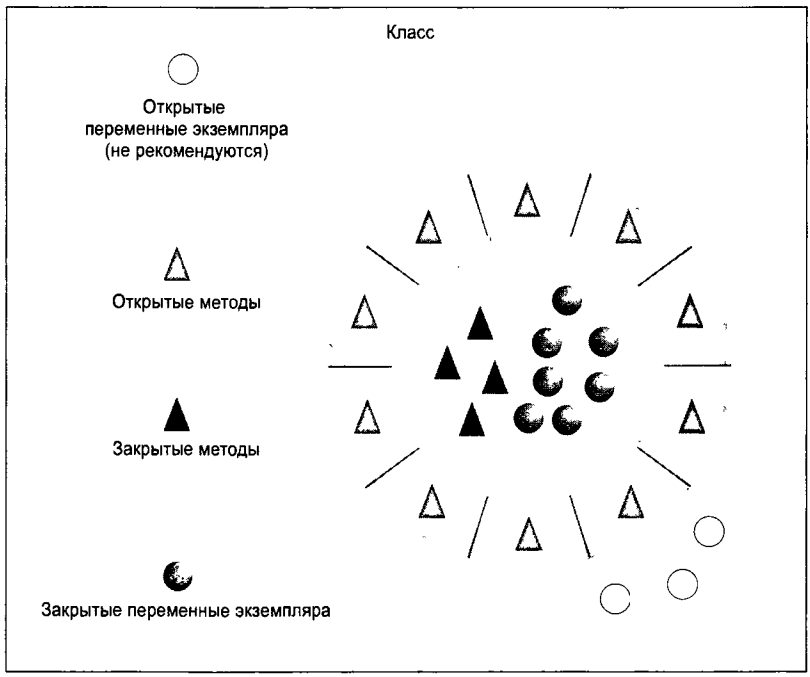


Рис. 2.1. Инкапсуляция: открытые методы можно использовать для защиты закрытых данных

Наследование

Процесс, в результате которого один объект получает свойства другого, именуется *наследованием*. Это важно, поскольку наследование обеспечивает концепцию иерархической классификации. Как уже отмечалось, большинство сведений становится доступным для понимания благодаря иерархической (т.е. нисходящей) классификации. Например, золотистый ретривер — часть классификации *собака*, которая, в свою очередь, относится к классу *млекопитающие*, а тот — к еще большему классу *животных*. Без использования иерархий каждый объект должен был бы явно определять все свои характеристики. Однако благодаря наследованию объект должен определять только те из них, которые делают его уникальным внутри класса. Объект может наследовать общие атрибуты от своего родительского объекта. Таким образом, механизм наследования обеспечивает возможность того, чтобы один объект был особым случаем более общего случая. Рассмотрим этот процесс подробнее.

Как правило, большинство людей воспринимают окружающий мир в виде иерархически связанных между собой объектов, подобных животным, млекопитающим и собакам.

Если требуется привести абстрактное описание животных, можно сказать, что они обладают определенными атрибутами, такими как размеры, уровень интеллекта и тип скелета. Животным присущи также определенные особенности поведения: они едят, дышат и спят. Приведенное описание атрибутов и поведения — определение *класса* животных.

Если бы требовалось описать более конкретный класс животных, например млекопитающих, нужно было бы указать более конкретные атрибуты, такие

как тип зубов и молочных желез. Это определение называют *подклассом* животных, которые относятся к *суперклассу* (родительскому классу) млекопитающих.

Поскольку млекопитающие — всего лишь более точно определенные животные, они *наследуют* все атрибуты животных. Подкласс нижнего уровня *иерархии классов* наследует все атрибуты каждого из его родительских классов (рис. 2.2).

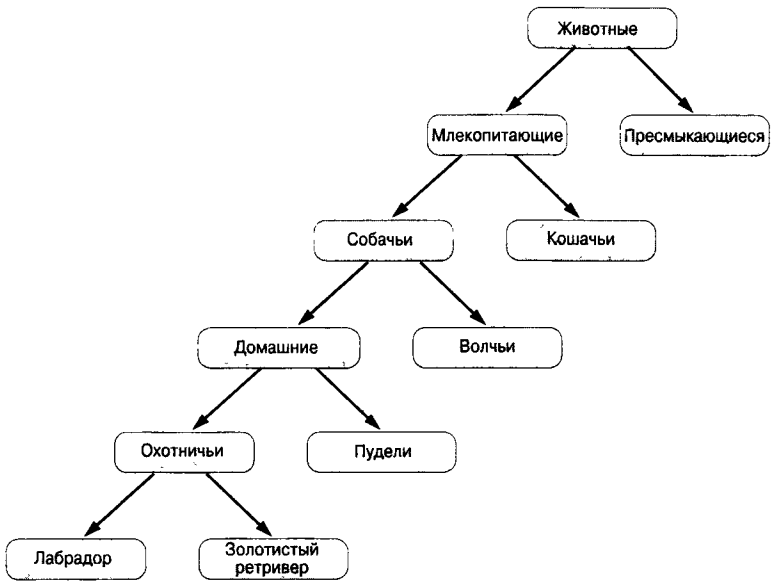


Рис. 2.2. Иерархия классов животных

Наследование связано также с инкапсуляцией. Если данный класс инкапсулирует определенные атрибуты, то любой его подкласс будет иметь эти же атрибуты *плюс* любые дополнительные атрибуты, являющиеся составной частью его специализации (рис. 2.3). Эта ключевая концепция позволяет сложности объектно-ориентированных программ возрастать в арифметической, а не геометрической прогрессии. Новый подкласс наследует атрибуты всех своих родительских классов. Поэтому он не содержит непредсказуемых взаимодействий с большей частью остального кода системы.

Полиморфизм

Полиморфизм (от греч. "много форм") — свойство, которое позволяет использовать один и тот же интерфейс для общего класса действий. Конкретное действие определяется конкретным характером ситуации. Рассмотрим стек (представляющий собой список типа "последним вошел, первым вышел"). Предположим, программе требуются стеки трех типов: для целочисленных значений, для значений с плавающей точкой и для символов. Алгоритм реализации каждого из этих стеков остается неизменным, несмотря на различие хранящихся в них данных. В не объектно-ориентированном языке пришлось бы создавать три различных набора подпрограмм стека, каждый из которых должен был бы иметь отдельное имя. Однако в Java, благодаря полиморфизму, можно определить общий набор подпрограмм стека, использующих одни и те же имена.

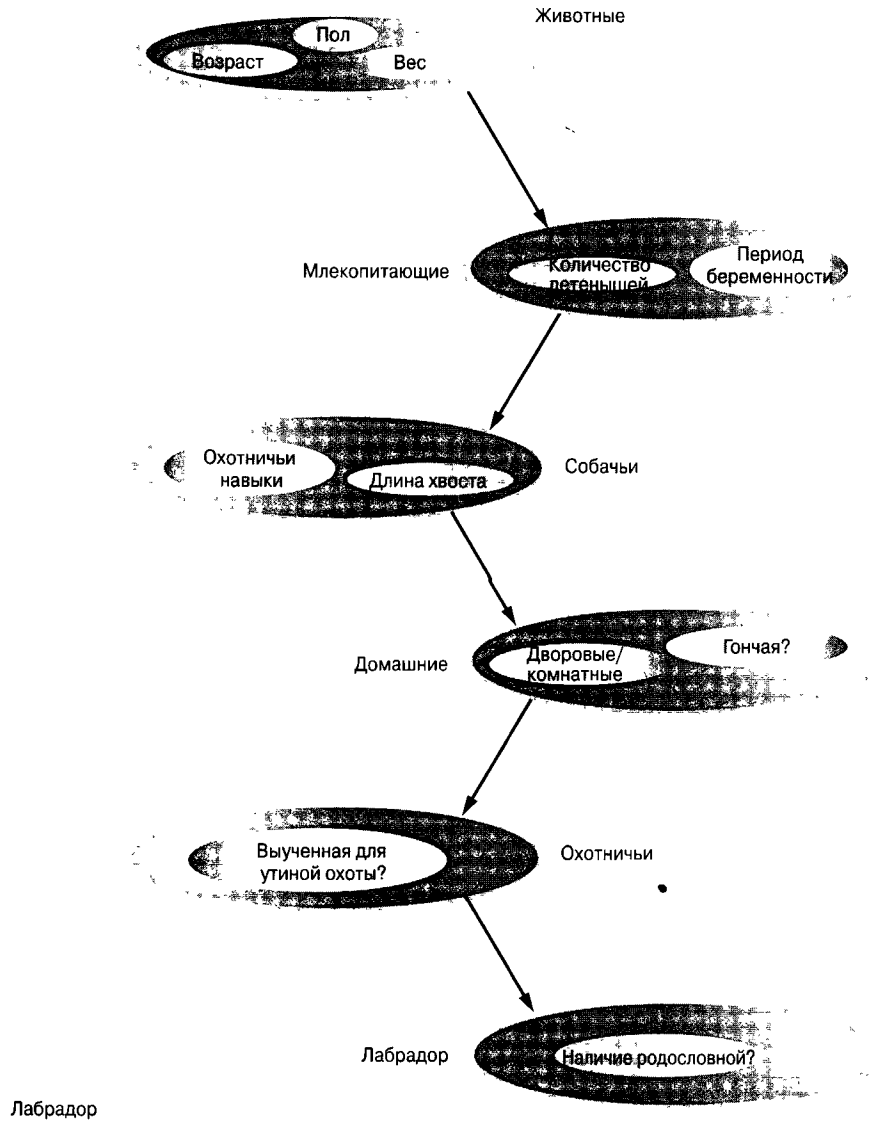


Рис. 2.3. Лабрадор полностью наследует инкапсулированные свойства всех родительских классов

В более общем виде концепцию полиморфизма часто выражают фразой "один интерфейс, несколько методов". Это означает, что можно спроектировать общий интерфейс для группы связанных между собой действий. Такой подход позволяет уменьшить сложность программы, поскольку один и тот же интерфейс использу-

ется для указания *общего класса действий*. Выбор же *конкретного действия* (т.е. метода), применимого к каждой ситуации, — задача компилятора. Программисту не нужно осуществлять этот выбор вручную. Необходимо лишь помнить об общем интерфейсе и применять его.

Если продолжить аналогию с собаками, можно сказать, что собачье обоняние — полиморфное свойство. Если собака ощутит запах кошки, она залает и погонится за ней. Если собака ощутит запах своего корма, у нее начнется слюноотделение, и она поспешит к своей миске. В обеих ситуациях действует одно и то же чувство обоняния. Различие в том, что издает запах, т.е. в типе данных, воздействующих на нос собаки! Эту же общую концепцию можно реализовать в Java применительно к методам внутри программы.

Совместное использование полиморфизма, инкапсуляции и наследования

При правильном совместном использовании полиморфизм, инкапсуляция и наследование создают среду программирования, которая поддерживает разработку более устойчивых и масштабируемых программ, чем в случае применения модели, ориентированной на процессы. Тщательно спроектированная иерархия классов — основа многократного использования кода, на разработку и проверку которого были затрачены время и усилия. Инкапсуляция позволяет возвращаться к ранее созданным реализациям, не разрушая код, зависящий от открытого интерфейса применяемых в приложении классов. Полиморфизм позволяет создавать понятный, чувствительный, удобочитаемый и устойчивый код.

Из приведенных ранее примеров реального мира пример с автомобилем более полно иллюстрирует возможности объектно-ориентированного проектирования. Пример с собаками хорошо подходит для рассмотрения его с точки зрения наследования, но автомобили имеют больше общего с программами. Садясь за руль различных типов (подклассов) автомобилей, все водители используют наследование. Независимо от того, является ли автомобиль школьным автобусом, легковым Мерседесом, Порше или семейным микроавтобусом, все водители смогут легко найти руль, тормоза, педаль акселератора и пользоваться ими. Немного повозившись с рычагом переключения передач, большинство людей могут даже оценить различия между ручной и автоматической коробками передач; это становится возможным благодаря получению четкого представления об общем родительском классе этих объектов — системе передач.

В процессе использования автомобилей люди постоянно взаимодействуют с их инкапсулированными характеристиками. Педали тормоза и газа скрывают невероятную сложность соответствующих объектов за настолько простым интерфейсом, что управлять этими объектами можно простым нажатием ступней педали! Конкретная реализация двигателя, тип тормозов и размер шин не оказывают никакого влияния на способ взаимодействия с определением класса педалей.

Последний атрибут, полиморфизм, четко отражает способность компаний-изготовителей автомобилей предлагать широкое множество вариантов, по сути, одного и того же средства передвижения. Например, на автомобиле могут быть установлены система тормозов с защитой от блокировки или традиционные тормоза, рулевая система с гидроусилителем или с реечной передачей и 4-, 6- или 8-цилиндровые двигатели. В любом случае нужно будет жать на педаль тормоза, чтобы остановиться, вращать руль, чтобы повернуть, и жать на педаль акселератора, чтобы автомобиль двигался. Один и тот же интерфейс может применяться для управления множеством различных реализаций.

Как видите, именно совместное применение инкапсуляции, наследования и полиморфизма позволяет преобразовать отдельные детали в объект, который мы называем автомобилем. Сказанное применимо также к компьютерным программам. За счет применения объектно-ориентированных принципов различные части сложной программы могут быть собраны воедино, образуя надежную, пригодную для обслуживания программу.

Как было отмечено в начале этого раздела, каждая программа Java является объектно-ориентированной. Или, точнее, каждая программа Java использует инкапсуляцию, наследование и полиморфизм. Хотя на первый взгляд может показаться, что не все эти свойства используются в приведенных в остальной части этой главы и нескольких последующих главах коротких примерах, тем не менее они в них присутствуют. Как вы вскоре убедитесь, многие функции, предоставляемые языком Java, являются составной частью его встроенных библиотек классов, в которых интенсивно применяются инкапсуляция, наследование и полиморфизм.

Первый пример простой программы

Теперь, когда мы ознакомились с исходными положениями объектно-ориентированного фундамента Java, рассмотрим несколько реальных программ, написанных на этом языке. Начнем с компиляции и запуска представленного в этом разделе короткого примера программы. Вы убедитесь, что эта задача более трудоемкая, чем может показаться.

```
/*
    Это простая программа Java.
    Назовите этот файл "Example.java".
*/

class Example {
    // Программа начинается с обращения к main().
    public static void main(String args[]) {
        System.out.println("Простая программа Java.");
    }
}
```

На заметку! Последующее описание соответствует стандарту комплекта разработчика Java SE 7 Developer's Kit (JDK 7), поставляемому компанией Oracle. При использовании другой среды разработки Java, компиляция и выполнение программ могут потребовать выполнения другой процедуры. Для получения необходимых сведений обратитесь к документации используемого компилятора.

Ввод кода программы

Для большинства языков программирования имя файла, который содержит исходный код программы, не имеет значения. Однако в Java это не так. Прежде всего, следует твердо усвоить, что присваиваемое исходному файлу имя очень важно. В данном случае именем исходного файла должно быть `Example.java`. И вот почему.

В Java исходный файл официально называется *модулем компиляции*. Он представляет собой текстовый файл (между прочим), который содержит определения одного или нескольких классов. (Мы пока будем использовать файлы исходного

кода, содержащие только один класс.) Компилятор Java требует, чтобы исходный файл имел расширение .java.

Как видно из кода программы, именем определенного программой класса является также Example. И это не случайно. В Java весь код должен размещаться внутри класса. В соответствии с принятым соглашением имя этого главного класса должно совпадать с именем файла, содержащего программу. Необходимо также, чтобы написание имени файла соответствовало имени класса, включая строчные и прописные буквы.

Это обусловлено тем, что код Java чувствителен к регистру символов. Пока что соглашение о соответствии имен файлов и имен классов может казаться произвольным. Однако оно упрощает поддержку и организацию программ.

Компиляция программы

Чтобы скомпилировать программу Example, запустите компилятор (javac), указав имя исходного файла в командной строке.

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

Компилятор javac создаст файл Example.class, содержащий версию кода виртуальной машины программы. Как мы уже отмечали, код виртуальной машины Java — это промежуточное представление программы, содержащее инструкции, которые будет выполнять виртуальная машина Java. Следовательно, результат работы компилятора javac не является непосредственно исполняемым кодом.

Чтобы действительно выполнить программу, необходимо воспользоваться программой запуска приложений Java, которая носит имя java. При этом ей потребуется передать имя класса Example в качестве аргумента командной строки, как показано в следующем примере.

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

При выполнении программы на экране отобразится следующий вывод.

```
Простая программа Java.
```

В процессе компиляции исходного кода каждый отдельный класс помещается в собственный выходной файл, названный по имени класса и получающий расширение .class. Именно поэтому исходным файлам Java целесообразно присваивать имена, совпадающие с именами классов, которые они содержат, — имя исходного файла будет совпадать с именем файла с расширением .class. При запуске java описанным способом в командной строке в действительности указывают имя класса, который нужно выполнить. Программа будет автоматически искать файл с указанным именем и расширением .class. Если программа найдет файл, она выполнит код, содержащийся в указанном классе.

Более подробное рассмотрение первого примера программы

Хотя программа Example.java небольшая, с ней связано несколько важных особенностей, характерных для всех программ Java. Давайте рассмотрим каждую часть этой программы более подробно. Программа начинается со следующих срок.

```
/*
    Это простая программа Java.
```

```
    Назовите этот файл "Example.java".
*/
```

Этот фрагмент кода — *комментарий*. Подобно большинству других языков программирования, Java позволяет вставлять примечания в исходный файл программы. Компилятор игнорирует содержимое комментариев. Эти комментарии описывают или поясняют действия программы для любого, кто просматривает исходный код. В данном случае комментарий описывает программу и напоминает, что исходный файл должен быть назван `Example.java`. Конечно, в реальных приложениях комментарии служат, главным образом, для пояснения работы отдельных частей программы или действий, выполняемых отдельной функцией.

В Java поддерживается три стиля комментариев. Комментарий, приведенный в начале программы, называют *многострочным комментарием*. Этот тип комментария должен начинаться с символов `/*` и заканчиваться символами `*/`. Весь текст, помещенный между этими двумя символами, компилятором игнорируется. Как следует из его названия, многострочный комментарий может содержать несколько строк.

Следующая строка программы имеет такой вид.

```
class Example {
```

В этой строке ключевое слово `class` используется для объявления о том, что выполняется определение нового класса. `Example` — это *идентификатор*, являющийся именем класса. Все определение класса, в том числе его членов, будет размещаться между открывающей (`{`) и закрывающей (`}`) фигурными скобками. Пока мы не станем подробно останавливаться на рассмотрении особенностей реализации класса. Отметим только, что в среде Java все действия программы осуществляются внутри класса. В этом состоит одна из причин того, что все программы Java (по крайней мере, частично) являются объектно-ориентированными.

Следующая строка программы — *однострочный комментарий*.

```
// Программа начинается с обращения к main().
```

Это второй тип комментариев, поддерживаемый языком Java. *Однострочный комментарий* начинается с символов `//` и завершается концом строки. Как правило, программисты многострочные комментарии используют для вставки длинных примечаний, а однострочные — для коротких, построчных описаний. Третий тип комментариев — *комментарий документации* - будет рассмотрен далее в этой главе в разделе "Комментарии".

Следующая строка кода выглядит так.

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

Она начинается с метода `main()`. Как видно из предшествующего ей комментария, выполнение программы начинается с этой строки. Выполнение всех приложений Java начинается с вызова метода `main()`. Пока мы не можем раскрыть смысл каждого элемента этой строки подробно, поскольку для этого требуется четкое представление о подходе Java к инкапсуляции. Однако поскольку эта строка кода присутствует в большинстве примеров этой части книги, давайте кратко рассмотрим ее.

Ключевое слово `public` — *модификатор доступа*, который позволяет программисту управлять видимостью членов класса. Когда члену класса предшествует ключевое слово `public`, этот член доступен коду, расположенному вне класса, в котором определен данный член. (Противоположное ему по действию ключевое слово — `private`, которое препятствует использованию члена класса кодом, определенным вне его класса.) В данном случае метод `main()` должен быть опреде-

лен как public, поскольку при запуске программы он должен вызываться кодом, определенным вне его класса. Ключевое слово static позволяет вызывать метод main() без конкретизации экземпляра класса. Это необходимо потому, что метод main() вызывается виртуальной машиной Java до создания каких-либо объектов. Ключевое слово void просто сообщает компилятору, что метод main() не возвращает никаких значений. Как будет показано в дальнейшем, методы могут также возвращать значения. Не беспокойтесь, если все сказанное кажется несколько сложным. Все эти концепции подробно рассматриваются в последующих главах.

Как было отмечено, main() — это метод, вызываемый при запуске приложений Java. Не забывайте, что язык Java чувствителен к регистру символов. Следовательно, строка Main не эквивалентна строке main. Важно понимать, что компилятор Java будет выполнять компиляцию классов, не содержащих метод main(). Но программа запуска приложений (java) не имеет средств запуска таких классов. Поэтому, если бы вместо main мы ввели Main, компилятор все равно выполнил бы компиляцию программы. Однако программа запуска приложений java сообщила бы об ошибке, поскольку не смогла бы найти метод main().

Для передачи любой информации, необходимой методу, служат переменные, указываемые в скобках, которые следуют за именем метода. Эти переменные называют *параметрами*. Если для данного метода никакие параметры не требуются, следует указывать пустые скобки. Метод main() содержит только один параметр, но достаточно сложный. Часть String args[] объявляет параметр args, который представляет собой массив экземпляров класса String. (*Массивы* — это коллекции аналогичных объектов.) Объекты типа String хранят символьные строки. В данном случае параметр args принимает любые аргументы командной строки, присутствующие во время выполнения программы. В данной программе эта информация не используется, но в других программах, рассмотренных позже, она будет применяться.

Последним элементом строки является символ фигурной скобки ({). Он обозначает начало тела метода main(). Весь код, образующий метод, будет располагаться между открывающей и закрывающей фигурными скобками метода.

Еще один важный момент: метод main() служит всего лишь началом программы. Сложная программа может включать десятки классов, только один из которых должен содержать метод main(), чтобы выполнение было возможным. Кроме того, в некоторых случаях метод main() вам не нужен вообще, например, при создании аплетов. Эти программы Java, внедренные в веб-браузеры, не используют метод main(), поскольку в них применяются другие средства запуска выполнения.

Следующая строка кода приведена ниже. Обратите внимание на то, что она помещена внутри метода main().

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

Эта строка выводит на экран строку текста "Простая программа Java.", за которой следует новая строка. В действительности вывод выполняется встроенным методом println(). В данном случае метод println() отображает переданную ему строку. Как будет показано, этот метод можно применять для отображения и других типов информации. Строка начинается с идентификатора потока System.out. Хотя он слишком сложный, чтобы его можно было подробно пояснить на данном этапе, если говорить вкратце, System — это предопределенный класс, который предоставляет доступ к системе, а его переменная-член out — выходной поток, связанный с консолью.

Как легко догадаться, в реальных программах Java консольный вывод-ввод используется не очень часто. Поскольку многие современные компьютерные среды по своей природе являются оконными и графическими, в большинстве случаев консольный ввод-вывод применяется в простых служебных и демонстрационных

программах. Позднее мы рассмотрим другие способы вывода в языке Java, а пока продолжим применять методы консольного ввода-вывода.

Обратите внимание: оператор, использующий метод `println()`, завершается точкой с запятой. В Java все операторы заканчиваются этим символом. Причина отсутствия точки с запятой в конце остальных строк программы в том, что с технической точки зрения они не являются операторами.

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

Второй пример короткой программы

Вероятно, ни одна другая концепция не является для языка программирования столь важной, как концепция переменных. Как вы, вероятно, знаете, *переменная* — это именованная ячейка памяти, которой может быть присвоено значение внутри программы. Во время выполнения программы значение переменной может изменяться. Следующая программа демонстрирует способы объявления переменной и присвоения ей значения. Она иллюстрирует также некоторые новые аспекты консольного вывода. Как следует из комментариев в начале программы, этому файлу нужно присвоить имя `Example2.java`.

```
/*
    Это еще один короткий пример.
    Назовите этот файл "Example2.java".
*/

class Example2 {
    public static void main(String args[]) {
        int num;   // эта строка объявляет переменную по имени num

        num = 100; // эта строка присваивает переменной num значение 100

        System.out.println("Это переменная num: " + num);

        num = num * 2;

        System.out.print("Значение переменной num * 2 равно ");
        System.out.println(num);
    }
}
```

При запуске этой программы на экране отобразится следующий вывод.

```
Это переменная num: 100
Значение переменной num * 2 равно 200
```

Рассмотрим создание этого вывода подробнее. Первая строка этой программы еще не знакома читателю.

```
int num;   // эта строка объявляет переменную по имени num
```

Она объявляет целочисленную переменную num. Java (подобно большинству других языков) требует, чтобы переменные были объявлены до их использования.

Ниже приведена общая форма объявления переменных.

```
тип имя_переменной;
```

В этом объявлении *тип* указывает тип объявляемой переменной, а *имя_переменной* — имя переменной. Если нужно объявить несколько переменных заданного типа, можно использовать разделенный запятыми список имен пере-

менных. Java определяет несколько типов данных, в том числе целочисленный, символьный и с плавающей точкой. Ключевое слово int указывает целочисленный тип. В приведенном примере программы строка

```
num = 100; // эта строка присваивает переменной num значение, равное 100
```

присваивает переменной num значение 100. В Java оператором присваивания служит одиночный знак равенства.

Следующая строка кода выводит значение переменной num, которому предшествует текстовая строка "Это переменная num:".

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

В этом операторе знак "плюс" вызывает дописывание значения переменной num в конец предшествующей строки и вывод результирующей строки. (В действительности значение переменной num вначале преобразуется из целочисленного в строковый эквивалент, а затем объединяется с предшествующей строкой. Подробнее этот процесс описан в последующих разделах книги.) Этот подход можно обобщить. Используя оператор +, внутри одного вызова метода println() можно объединять необходимое количество строк.

Следующая строка кода присваивает переменной num значение этой переменной, умноженное на 2. Как и в большинстве других языков, в Java оператор * служит для указания операции умножения. После выполнения этой строки кода переменная num будет содержать значение 200.

Следующие две строки программы выглядят так.

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

В них выполняется несколько новых действий. Метод print() используется для отображения строки "Значение переменной num * 2 равно". За этой строкой *не* следует символ новой строки. Таким образом, следующий вывод будет отображаться в той же строке. Метод print() подобен методу println(), за исключением того, что после каждого вызова он не создает символ новой строки. Теперь рассмотрим вызов метода println(). Обратите внимание на то, что имя переменной num используется само по себе. Методы print() и println() могут применяться для вывода значений любых встроенных типов Java.

Два управляющих оператора

Хотя управляющие операторы подробно рассматриваются в главе 5, здесь мы кратко рассмотрим два управляющих оператора, чтобы их можно было использовать в примерах программ, приведенных в главах 3 и 4. Кроме того, они послужат хорошей иллюстрацией важного аспекта Java – блоков кода.

Оператор if

Работа оператора if в Java во многом аналогична работе оператора IF любого другого языка. Более того, его синтаксис идентичен синтаксису операторов if в языках C, C++ и C#. Простейшая форма этого оператора выглядит следующим образом.

```
if(условие) оператор;
```

Здесь *условие* — булево выражение. Если *условие* истинно, оператор выполняется. Если *условие* ложно, оператор игнорируется. Рассмотрим следующий пример.

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

В данном случае, если переменная num содержит значение меньше 100, условное выражение истинно и программа выполнит метод println(). Если переменная num содержит значение, которое больше или равно 100, программа пропустит вызов метода println().

Как будет показано в главе 4, в языке Java определен полный набор операторов сравнения, которые могут быть использованы в условном выражении. Некоторые из них перечислены в табл. 2.1.

Таблица 2.1. Некоторые операторы сравнения

Оператор	Значение
<	Меньше
>	Больше
==	Равно

Обратите внимание: символом проверки равенства служит двойной знак равенства.

Ниже приведен пример программы, иллюстрирующий применение оператора if.

```
/*
    Демонстрирует применение 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("x меньше y");

        x = x * 2;
        if(x == y) System.out.println("x теперь равна y");

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

        // этот оператор не будет ничего отображать
        if(x == y) System.out.println("вы не увидите это");
    }
}
```

Эта программа создает следующий вывод.

```
x меньше y
x теперь равна y
x теперь больше y
```

Обратите внимание еще на одну особенность этой программы. Строка

```
int x, y;
```

объявляет две переменные x и y, используя при этом разделенный запятой список.

Цикл for

Как вам, возможно, известно из уже имеющегося опыта программирования, операторы цикла — важная составная часть практически любого языка программирования. Язык Java — не исключение в этом отношении. Фактически, как будет показано в главе 5, Java поддерживает обширный набор операторов цикла. И, вероятно, наиболее универсальный из них — цикл for. Простейшая форма этого цикла имеет такой вид.

```
for(начальное_значение; условие; приращение) оператор;
```

В этой наиболее часто встречающейся форме параметр *начальное_значение* определяет начальное значение управляющей переменной цикла. *Условие* — это булево выражение, которое проверяет значение управляющей переменной цикла. Если результат проверки условия истинен, выполнение цикла for продолжается. Если результат проверки ложен, выполнение цикла прекращается. Выражение *приращение* определяет изменение управляющей переменной на каждой итерации цикла. Ниже показан пример короткой программы, иллюстрирующий применение цикла for.

```
/*
    Демонстрирует применение 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);
    }
}
```

Эта программа создает следующий 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 этом примере x — управляющая переменная цикла. В части инициализации цикла ей присвоено начальное значение, равное нулю. В начале каждой итерации (включая первую) выполняется проверка условия x<10. Если результат этой проверки истинен, программа выполняет метод println(), а затем итерационную часть цикла. Процесс продолжается до тех пор, пока результат проверки условия не станет ложным.

Следует отметить, что в профессионально написанных программах Java вы почти никогда не встретите итерационную часть цикла в том виде, какой она имеет в приведенном примере. Иными словами, операторы вроде следующего встречаются весьма редко.

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

Это объясняется тем, что Java предоставляет специальный, более эффективный оператор инкремента значения ++ (два последовательных знака "плюс"). Оператор инкремента значения увеличивает значение операнда на единицу. Используя этот оператор, предшествующее выражение можно было бы переписать следующим образом.

```
x++;
```

Таким образом, как правило, цикл for предшествующей программы будет иметь примерно такой вид.

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

Можете проверить выполнение этого цикла и убедиться, что он работает точно так же, как в предшествующем примере.

Язык Java предоставляет также оператор декремента значений --. Этот оператор уменьшает значение операнда на единицу.

Использование блоков кода

Язык Java позволяет группировать два и более операторов в *блоки кода*. Для этого операторы заключают в фигурные скобки. Сразу после создания блок кода становится логическим модулем, который можно использовать в тех же местах, что и отдельный оператор. Например, блок может служить в качестве цели для операторов if и for. Рассмотрим следующий оператор if.

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

В этом примере, если значение переменной x меньше y, программа выполнит оба оператора, расположенные внутри блока. Таким образом, оба оператора внутри блока образуют логический модуль, и выполнение одного оператора невозможно без одновременного выполнения и второго. Основная идея этого подхода в том, что во всех случаях, когда требуется логически связать два или более операторов, используется блок.

Рассмотрим еще один пример. В следующей программе блок кода использован в качестве целевого модуля цикла for.

```
/*
    Демонстрирует использование блока кода.

    Назовите этот файл "BlockTest.java"
*/
class BlockTest {
    public static void main(String args[]) {
        int x, y;

        y = 20;

        // целевой модуль этого цикла - блок
        for(x = 0; x<10; x++) {
            System.out.println("Значение x: " + x);
            System.out.println("Значение y: " + y);
            y = y - 2;
        }
```

```
    }
}
```

Эта программа создает 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 данном случае цель цикла `for` — блок кода, а не единственный оператор. Таким образом, при каждой итерации цикла программа будет выполнять три оператора, помещенные внутрь блока. Естественно, об этом свидетельствует и вывод программы.

Как будет показано в последующих главах книги, блоки кода обладают дополнительными свойствами и областями применения. Однако их основное назначение — создание логически неразрывных модулей кода.

Вопросы лексики

Теперь, когда читатели ознакомились с несколькими короткими программами Java, пора более формально описать основные элементы языка. Программы на языке Java представляют собой коллекцию пробелов, идентификаторов, литералов, комментариев, операторов, разделителей и ключевых слов. Операторы рассматриваются в следующей главе, а остальные элементы описаны в последующих разделах данной главы.

Отступ

Java — язык свободной формы. Это означает, что при написании программы не нужно следовать никаким специальным правилам в отношении отступов. Например, программу `Example` можно было бы записать в виде одной строки или любым другим способом. Единственное обязательное требование — наличие, по меньшей мере, одного пробела между всеми лексемами, которые еще не разграничены оператором или разделителем. В языке Java отступами являются символы пробела, табуляции или символы новой строки.

Идентификаторы

Для именования классов, методов и переменных используются идентификаторы. Идентификатором может служить любая последовательность строчных и прописных букв, цифр или символов подчеркивания и доллара. (Для общего использования символ доллара не предназначается.) Идентификаторы не должны начинаться с цифры, чтобы компилятор не путал их с числовыми константами. Повторим еще раз, что язык Java чувствителен к регистру символов, и поэтому `VALUE` и `Value` — различные идентификаторы. Ниже приведено несколько примеров допустимых идентификаторов.

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

Следующие идентификаторы недопустимы.

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

Литералы

В Java постоянное значение задается его *литеральным* представлением. Например, ниже показано несколько литералов.

```
100   98.6   'X'   "This is a test"
```

Первый литерал указывает целочисленное значение, следующий — значение с плавающей точкой, третий — символьную константу, а последний — строковое значение. Литерал можно использовать везде, где допустимо использование значений данного типа.

Комментарии

Как уже было отмечено, в Java определено три типа комментариев. Два из них мы уже встречали: однострочные и многострочные. Третий тип называется *комментарием документации*. Этот тип комментариев используется для создания файла HTML документации программы. Комментарий документации начинается с символов `/**` и заканчивается символами `*/`. Подробнее комментарии документации описаны в приложении.

Разделители

Java допускает применение нескольких символов в качестве разделителей. Чаще всего в качестве разделителя используется точка с запятой. Как вы уже видели, она применяется для завершения строк операторов. Допустимые символы-разделители описаны в табл. 2.2.

Таблица 2.2. Допустимые символы-разделители

Символ	Название	Назначение
( )	Круглые скобки	Используются для передачи списков параметров в определениях и вызовах методов. Их применяют также для определения приоритета в выражениях, выражений в управляющих операторах и преобразования типов

Окончание табл. 2.2

Символ	Название	Назначение
{ }	Фигурные скобки	Используются для указания значений автоматически инициализируемых массивов. Их применяют также для определения блоков кода, классов, методов и локальных областей видимости
[ ]	Квадратные скобки	Используются для объявления типов массивов, а также при обращении к значениям массивов
;	Точка с запятой	Завершает операторы
,	Запятая	Разделяет последовательные идентификаторы в объявлениях переменных. Этот символ-разделитель используют также для создания цепочек операторов внутри оператора `for`
.	Точка	Используется для отделения имен пакетов от имен вложенных пакетов и классов, а также для отделения переменной или метода от ссылочной переменной

Ключевые слова Java

В настоящее время в языке Java определено 50 ключевых слов (табл. 2.3). Эти ключевые слова, в сочетании с синтаксисом операторов и разделителями, образуют основу языка Java. Их нельзя использовать ни в качестве идентификаторов, ни для имен переменных, классов или методов.

Ключевые слова `const` и `goto` зарезервированы, но не используются. На начальном этапе разработки языка Java еще несколько ключевых слов были зарезервированы для возможного использования в будущем. Однако в настоящее время спецификация языка Java определяет только ключевые слова, перечисленные в табл. 2.3.

Таблица 2.3. Ключевые слова Java

`abstract`	`continue`	`for`	`new`	`switch`
`assert`	`default`	`goto`	`package`	`synchronized`
`boolean`	`do`	`if`	`private`	`this`
`break`	`double`	`implements`	`protected`	`throw`
`byte`	`else`	`import`	`public`	`throws`
`case`	`enum`	`instanceof`	`return`	`transient`
`catch`	`extends`	`int`	`short`	`try`
`char`	`final`	`interface`	`static`	`void`
`class`	`finally`	`long`	`strictfp`	`volatile`
`const`	`float`	`native`	`super`	`while`

Кроме ключевых слов, в Java зарезервированы также слова `true`, `false` и `null`. Они представляют значения, определенные спецификацией языка Java. Их нельзя использовать в качестве имен переменных, классов и т.п.

Библиотеки классов Java

В приведенных в этой главе примерах программ использовались два встроенных метода Java: `println()` и `print()`. Как уже говорилось, эти методы являются членами класса `System`, который представляет собой стандартный класс Java, автоматически включаемый в программы. В более широком смысле среда Java построена на основе нескольких встроенных библиотек классов, содержащих множество встроенных методов, которые предоставляют поддержку выполнения таких задач, как ввод-вывод, обработка строк, сетевая обработка и обработка графики. Стандартные классы предлагают также поддержку оконного вывода. Таким образом, в целом среда Java представляет собой сочетание самого языка Java и его стандартных классов. Как будет показано, библиотеки классов предоставляют большую часть функциональных возможностей, обеспечиваемых средой Java. Действительно, в определенной степени стать программистом на Java означает научиться использовать стандартные классы Java. В главах этой части описание различных элементов стандартных библиотечных классов и методов приводится по мере необходимости, а в части II библиотеки классов описаны более подробно.

ГЛАВА 3

Типы данных, переменные и массивы

В этой главе рассмотрены три наиболее важных элемента Java: типы данных, переменные и массивы. Как и все современные языки программирования, Java поддерживает несколько типов данных. Эти типы можно применять для объявления переменных и создания массивов. Как будет показано, подход к использованию этих компонентов, примененный в Java, прост, эффективен и целостен.

Java — строго типизированный язык

Прежде всего, важно уяснить, что Java — строго типизированный язык. Действительно, в определенной степени безопасность и надежность программ Java обусловлены именно этим обстоятельством. Давайте разберемся, что это означает. Во-первых, каждая переменная обладает типом, каждое выражение имеет тип и каждый тип строго определен. Во-вторых, все присваивания, как явные, так и за счет передачи параметров в вызовах методов, проверяются на соответствие типов. В Java отсутствуют какие-либо средства автоматического приведения или преобразования конфликтующих типов, как это имеет место в некоторых языках. Компилятор Java проверяет все выражения и параметры на предмет совместимости типов. Любые несоответствия типов являются ошибками, которые должны быть исправлены до завершения компиляции класса.

Элементарные типы

Язык Java определяет восемь *элементарных* типов данных: `byte`, `short`, `int`, `long`, `char`, `float`, `double` и `boolean`.

Элементарные типы называют также *простыми*, и в этой книге мы будем использовать оба эти термина. Элементарные типы можно разделить на четыре группы.

- Целые числа. Эта группа включает в себя типы `byte`, `short`, `int` и `long`, которые представляют точные целые числа со знаком.
- Числа с плавающей точкой. Эта группа включает в себя типы `float` и `double`, которые представляют числа, определенные с точностью до определенного десятичного знака.

- Символы. Эта группа включает в себя тип `char`, которая представляет символы символьного набора, такие как буквы и цифры.
- Булевы значения. Эта группа включает в себя тип `boolean` — специальный тип, предназначенный для представления значений типа "истинно/ложно".

Эти типы можно использовать в том виде, как они определены, или же для создания собственных типов классов. Таким образом, они служат основой для всех других типов данных, которые могут быть созданы.

Элементарные типы представляют одиночные значения, а не сложные объекты. Хотя во всех других отношениях Java — полностью объектно-ориентированный язык, элементарные типы данных таковыми не являются. Они аналогичны простым типам, которые можно встретить в большинстве других не объектно-ориентированных языков. Эта особенность обусловлена стремлением обеспечить максимальную эффективность. Превращение элементарных типов в объекты привело бы к слишком большому снижению производительности.

Элементарные типы определены так, чтобы они обладали явным диапазоном допустимых значений и математически строгим поведением. Языки вроде C и C++ допускают варьирование размеров целочисленных переменных в зависимости от требований среды исполнения. Однако Java отличается в этом отношении. В связи с требованием переносимости, предъявляемым к программам Java, все типы данных обладают строго определенным диапазоном допустимых значений. Например, независимо от конкретной платформы, значения типа `int` всегда являются 32-битовыми. Это позволяет создавать программы, которые гарантированно будут выполняться на любой машинной архитектуре *без специального переноса*. Хотя в некоторых средах строгое указание размера целых чисел может приводить к незначительному снижению производительности, оно абсолютно необходимо для обеспечения переносимости программ.

Рассмотрим каждый из типов данных.

Целочисленные значения

Язык Java определяет четыре целочисленных типа: `byte`, `short`, `int` и `long`. Все эти типы представляют значения со знаком — положительные и отрицательные. Java не поддерживает только положительные целочисленные значения без знака. Многие другие языки программирования поддерживают целочисленные значения как со знаком, так и без знака. Однако разработчики Java посчитали целочисленные значения без знака ненужными. В частности, они решили, что концепция *значений без знака* использовалась, в основном, для указания поведения *старшего бита*, который определяет *знак* целочисленного значения. Как будет показано в главе 4, в Java управление значением старшего бита осуществляется иначе — за счет применения специальной операции "сдвига вправо без учета знака". Тем самым потребность в целочисленном типе без знака была исключена.

Ширина целочисленного типа представляет не занимаемый объем памяти, а, скорее, *поведение*, определяемое им для переменных и выражений этого типа. Среда времени выполнения Java может использовать любой размер до тех пор, пока типы ведут себя объявленным образом. Как показано в табл. 3.1, ширина и диапазон допустимых значений этих целочисленных типов изменяются в широких пределах.

Таблица 3.1. Ширина и диапазон допустимых значений целочисленных типов

Имя	Ширина	Диапазон допустимых значений
`long`	64	от -9223372036854775808 до 9223372036854775807
`int`	32	от -2147483648 до 2147483647
`short`	16	от -32768 до 32767
`byte`	8	от -128 до 127

Теперь рассмотрим каждый из типов целочисленных значений.

Тип `byte`

Наименьший по размеру целочисленный тип — `byte`. Это 8-битовый тип со знаком и диапазоном допустимых значений от -128 до 127. Переменные типа `byte` особенно полезны при работе с потоком данных, поступающих из сети или файла. Они полезны также при манипулировании необработанными двоичными данными, которые могут не быть непосредственно совместимыми с другими встроенными типами Java.

Для объявления переменных типа `byte` служит ключевое слово `byte`. Например, в следующей строке объявлены две переменные типа `byte` — `b` и `c`.

```
byte b, c;
```

Тип `short`

Тип `short` — 16-битовый тип со знаком. Он имеет диапазон допустимых значений от -32768 до 32767. Вероятно, этот тип используется в Java наименее часто. Ниже приведено несколько примеров объявления переменных типа `short`.

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

Тип `int`

Наиболее часто используемым целочисленным типом является `int`. Это 32-битовый тип со знаком, который имеет диапазон допустимых значений от -2147483648 до 2147483647. Кроме других способов применения, переменные типа `int` зачастую используются для управления циклами и индексирования массивов. Хотя на первый взгляд может показаться, что использование типов `byte` или `short` эффективнее использования типа `int` в ситуациях, когда не требуется более широкий диапазон допустимых значений, предоставляемый последним, в действительности это не всегда так. Это обусловлено тем, что при указании значений типа `byte` и `short` в выражениях их тип повышается до типа `int` при вычислении выражения. (Повышение типа описано в этой главе позже.) Поэтому тип `int` зачастую наиболее подходит для работы с целочисленными значениями.

Тип `long`

Это 64-битовый тип со знаком, удобный в тех ситуациях, когда длина типа `int` недостаточна для хранения требуемого значения. Диапазон допустимых значений

типа long достаточно велик. Это делает его удобным для работы с большими целыми числами.

Например, ниже приведен пример программы, которая вычисляет количество миль, проходимых лучом света за указанное число дней.

```
// Вычисление расстояния, проходимого 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; // указание количества дней

        seconds = days * 24 * 60 * 60; // преобразование в секунды

        distance = lightspeed * seconds; // вычисление расстояния

        System.out.print("За " + days);
        System.out.print(" дней свет пройдет около ");
        System.out.println(distance + " миль.");
    }
}
```

Эта программа создает следующий вывод.

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

Очевидно, что результат не поместился бы в переменной типа int.

Типы с плавающей точкой

Числа с плавающей точкой, называемые также *действительными* числами, используются при вычислении выражений, которые требуют результата с точностью до десятичного знака. Например, вычисление квадратного корня или трансцендентных функций вроде синуса или косинуса приводит к результату, который требует применения типа с плавающей точкой. В Java реализован стандартный (в соответствии с IEEE-754) набор типов и операций с плавающей точкой. Существует два типа с плавающей точкой: float и double, которые соответственно представляют числа одинарной и двойной точности. Их ширина и области допустимых значений описаны в табл. 3.2.

Таблица 3.2. Ширина и диапазон допустимых значений типов с плавающей точкой

Имя	Ширина в битах	Приблизительный диапазон допустимых значений
double	64	от 4.9e-324 до 1.8e+308
float	32	от 1.4e-045 до 3.4e+038

Рассмотрим каждый из этих типов с плавающей точкой.

Тип `float`

Этот тип определяет *значение одинарной* точности, которое при хранении занимает 32 бит. В некоторых процессорах обработка значений одинарной точности выполняется быстрее и требует в два раза меньше памяти, чем обработка значений двойной точности, но в тех случаях, когда значения либо очень велики, либо очень малы, точность вычислений оказывается недостаточной. Переменные типа `float` удобны в тех случаях, когда требуется дробная часть значения без особой точности. Например, значение типа `float` может быть удобно для представления денежных сумм в долларах и центах. Ниже приведен пример объявлений переменных типа `float`.

```
float hightemp, lowtemp;
```

Тип `double`

Двойная точность, как следует из ключевого слова `double` (двойная), требует использования 64 бит для хранения значений. В действительности в некоторых современных процессорах, которые оптимизированы для выполнения математических вычислений с высокой скоростью, обработка значений двойной точности осуществляется быстрее, чем обработка значений одинарной точности. Все трансцендентные математические функции, такие как `sin()`, `cos()` и `sqrt()`, возвращают значения типа `double`. Применение типа `double` наиболее рационально, когда требуется сохранение точности множества последовательных вычислений или манипулирование большими числами.

Ниже приведен пример короткой программы, в которой переменные типа `double` используются для вычисления площади круга.

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

        r = 10.8;        // радиус окружности
        pi = 3.1416;     // pi, приблизительное значение
        a = pi * r * r; // вычисление площади

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

Символы

В Java для хранения символов используется тип данных `char`. Однако программистам на C/C++ следует помнить, что тип `char` в Java не эквивалентен типу `char` в C или C++. В C/C++ `char` — это целочисленный тип, имеющий ширину 8 бит. В Java это *не так*. Вместо этого в нем для представления символов используется кодировка Unicode, определяющая международный набор символов, который может представлять все символы известных языках. Кодировка Unicode представляет собой унифицированный набор из десятков наборов символов, таких как латиница, греческий алфавит, арабский алфавит, кириллица, иврит, японские и тайские иероглифы, а также множество других. Поэтому для хранения этих символов требуется 16 бит. Таким образом, в Java тип `char` является 16-битовым. Диапазон допустимых значений этого типа — от 0 до 65536. Не существует отрицательных значений типа `char`. Стандартный

набор символов, известный как ASCII, содержит значения от 0 до 127, а расширенный 8-битовый набор символов, ISO-Latin-1, – значения от 0 до 255. Поскольку язык Java предназначен для обеспечения возможности создания программ, применимых во всем мире, использование кодировки Unicode для представления символов вполне обоснованно. Конечно, применение Unicode несколько неэффективно для таких языков, как английский, немецкий, испанский или французский, для представления символов которых вполне достаточно 8 бит. Но это та цена, которую приходится платить за переносимость программ во всемирном масштабе.

На заметку! Более подробно о кодировке Unicode см. на веб-сайте `http://www.unicode.org`.

Использование переменных типа `char` демонстрирует следующая программа.

```
// Демонстрация использования типа данных char.
class CharDemo {
    public static void main(String args[]) {
        char ch1, ch2;

        ch1 = 88; // код переменной X
        ch2 = 'Y';

        System.out.print("ch1 и ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

Эта программа отображает следующий вывод.

```
ch1 и ch2: X Y
```

Обратите внимание на то, что переменной `ch1` присвоено значение 88, являющееся значением ASCII (и Unicode), которое соответствует букве X. Как уже было сказано, набор символов ASCII занимает первые 127 значений набора символов Unicode. Поэтому все "старые трюки", применяемые при работе с символами в других языках, будут работать и в среде Java.

Хотя тип `char` был разработан для хранения символов Unicode, его можно также использовать как целочисленный тип, пригодный для выполнения арифметических операций.

Например, он позволяет выполнять сложение символов или уменьшать значение символьной переменной. Рассмотрим следующую программу.

```
// Символьные переменные ведут себя подобно целочисленным значениям.
class CharDemo2 {
    public static void main(String args[]) {
        char ch1;

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

        ch1++; // увеличение значения ch1 на единицу
        System.out.println("ch1 теперь " + ch1);
    }
}
```

Эта программа создает следующий вывод.

```
ch1 содержит X
ch1 теперь Y
```

Вначале программа присваивает переменной ch1 значение X. Затем она увеличивает значение переменной ch1 на единицу. В результате хранящееся в переменной значение становится буквой Y — следующим символом в последовательности ASCII (и Unicode).

На заметку! В формальной спецификации Java тип char упоминается как целочисленный, это значит, что он находится в той же общей категории, что и типы int, short, long и byte. Однако поскольку основное назначение типа char — представлять символы Unicode, он находится в собственной категории.

Булевы значения

Язык Java содержит элементарный тип, названный boolean, который предназначен для хранения логических значений. Переменные этого типа могут принимать только одно из двух возможных значений: true (истинно) или false (ложно). Этот тип возвращается всеми операторами сравнения, подобными a<b. Тип boolean обязателен для использования также в условных выражениях, которые управляют такими управляющими операторами, как if и for.

Следующая программа служит примером использования типа boolean.

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

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

        // значение типа boolean может управлять оператором if
        if(b) System.out.println("Это выполняется.");
        b = false;
        if(b) System.out.println("Это не выполняется.");

        // результат сравнения - значение типа boolean
        System.out.println("10 > 9 равно " + (10 > 9));
    }
}
```

Эта программа создает следующий вывод.

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

В приведенной программе особый интерес представляют три момента. Во-первых, как видите, при выводе значения типа boolean методом println() на экране отображается строка "true" или "false". Во-вторых, самого по себе значения переменной типа boolean достаточно для управления оператором if. Вовсе не обязательно записывать оператор if так, как показано ниже.

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

В-третьих, результат выполнения оператора сравнения, такого как <, — значение типа boolean. Именно поэтому выражение 10 > 9 приводит к отображению строки "true". Более того, выражение 10 > 9 должно быть заключено в допол-

нительный набор круглых скобок, поскольку оператор + обладает более высоким приоритетом, чем оператор >.

Более подробное рассмотрение литералов

Литералы были вскользь упомянуты в главе 2. Теперь, когда встроенные типы формально описаны, рассмотрим их подробнее.

Целочисленные литералы

Целочисленные значения — вероятно, наиболее часто используемый тип в типичной программе. Любое целочисленное значение является числовым литералом. Примерами могут служить значения 1, 2, 3 и 42. Все они — десятичные значения, описывающие числа с основанием 10. В числовых литералах могут использоваться еще два вида представления — *восьмеричное* (с основанием 8) и *шестнадцатеричное* (с основанием 16). В Java восьмеричные значения обозначаются ведущим нулем. Обычные десятичные числа не могут содержать ведущий нуль. Таким образом, внешне вполне допустимое значение 09 приведет к ошибке компиляции, поскольку 9 выходит за пределы диапазона от 0 до 7 допустимых восьмеричных значений. Чаще программисты используют шестнадцатеричное представление чисел, которое явно соответствует словам, размер которых равен 8, 16, 32 и 64 бит, составленным из 8-битовых блоков. Значения шестнадцатеричных констант обозначают ведущим нулем и символом x (`0x` или `0X`). Диапазон допустимых шестнадцатеричных цифр — от 0 до 15, поэтому цифры от 10 до 15 заменяют буквами от `A` до `F` (или от `a` до `f`).

Целочисленные литералы создают значение типа `int`, которое в Java является 32-битовым целочисленным значением. Поскольку Java — строго типизированный язык, может возникать вопрос, каким образом можно присваивать целочисленный литерал одному из других целочисленных типов Java, такому как `byte` или `long`, не вызывая при этом ошибку несоответствия типа. К счастью, с подобными ситуациями легко справиться. Когда значение литерала присваивается переменной типа `byte` или `short`, ошибка не происходит, если значение литерала находится в диапазоне допустимых значений этого типа. Кроме того, целочисленный литерал всегда можно присваивать переменной типа `long`. Однако чтобы указать литерал типа `long`, придется явно указать компилятору, что значение литерала имеет этот тип. Для этого к литералу дописывают строчную или прописную букву `L`. Например, `0x7fffffffffffffL`, или `9223372036854775807L`, — наибольший литерал типа `long`. Целочисленное значение можно присваивать типу `char`, если оно лежит в пределах допустимого диапазона этого типа.

Начиная с JDK 7 вы можете также определить целочисленные литералы, используя двоичную форму. Для этого перед значением используется префикс `0b` или `0B`. Например, следующий код определяет десятичное значение 10 с использованием двоичного литерала.

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

Наличие двоичных литералов облегчает также ввод значений, используемых как битовые маски. В таком случае десятичное (или шестнадцатеричное) представление значения числа не отображает визуально способ его использования, а двоичный литерал отображает.

Начиная с JDK 7 вы можете также внедрить в целочисленный литерал один или несколько символов подчеркивания. Это облегчает чтение больших целочисленных литералов. При компиляции символы подчеркивания в литерале игнорируются. Например, в данном случае

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

значением переменной x будет 123 456 789. Символы подчеркивания будут проигнорированы. Символы подчеркивания можно использовать только для разделения цифр. Они не могут располагаться в начале или в конце литерала, но вполне допустимо использование между двумя цифрами нескольких символов подчеркивания. Например, следующее вполне правильно.

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

Использование символов подчеркивания в целочисленном литерале особенно полезно при написании в коде таких элементов, как номера телефонов, идентификационные номера клиентов, номера частей и т.д. Они также полезны для визуальных группировок при определении двоичных литералов. Например, двоичные значения зачастую визуально группируются в блоки по четыре цифры, как показано далее.

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

Литералы с плавающей точкой

Числа с плавающей точкой представляют десятичные значения с дробной частью. Они могут быть выражены в стандартной или научной форме записи. Число в *стандартной форме записи* состоит из целого числа, за которым следуют десятичная точка и дробная часть. Например, `2.0`, `3.14159` и `0.6667` представляют допустимые числа с плавающей точкой в стандартной записи. *Научная форма записи* использует стандартную форму записи числа с плавающей точкой, к которой добавлен суффикс, указывающий степенную функцию числа 10, на которую нужно умножить данное число. Для указания экспоненциальной функции используют символ `E` или `e`, за которым следует десятичное число (положительное или отрицательное). Примерами могут служить `6.022E23`, `314159E-05` и `2e+100`.

По умолчанию в Java литералам с плавающей точкой присвоен тип `double`. Чтобы указать литерал типа `float`, к нему нужно дописать символ `F` или `f`. Литерал типа `double` можно также указать явно, дописывая к нему символ `D` или `d`. Но, естественно, это излишне. Используемый по умолчанию тип `double` занимает в памяти 64 бит, в то время как меньший тип `float` требует для хранения только 32 бит.

Шестнадцатеричные литералы с плавающей точкой также поддерживаются, но используются они редко. Они должны быть в форме, подобной экспоненциальному представлению, но с `P` или `p` вместо `E` или `e`. Например, `0x12.2P2` — вполне допустимый литерал с плавающей точкой. Значение после буквы `P` называется *двоичным порядком* (binary exponent) и указывает "степень числа два", на которое умножается число. Поэтому `0x12.2P2` представляет число 72,5.

Начиная с JDK 7 вы можете внедрить в литерал с плавающей точкой один или несколько символов подчеркивания. Это работает точно так же, как и для целочисленных литералов, которые были описаны выше. Символы подчеркивания облегчают чтение больших литералов с плавающей точкой. При компиляции символы подчеркивания игнорируются. Например, в данном случае

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

значением num будет 9 423 497 862,0. Символы подчеркивания будут проигнорированы. Как и в случае с целочисленными литералами, символы подчеркивания могут использоваться только для разделения цифр. Они не могут располагаться в начале или конце литерала, но вполне допустимо использование между двумя цифрами нескольких символов подчеркивания. Допустимо также использование символов подчеркивания в дробной части числа. Например, следующий код вполне правилен.

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

В данном случае дробная часть — .109.

Булевы литералы

Эти литералы очень просты. Тип Boolean может иметь только два значения: true и false. Эти значения не преобразуются ни в какие числовые представления. В Java литерал true не равен 1, а литерал false — 0. В Java логические литералы могут быть присвоены только тем переменным, которые объявлены как boolean, или использоваться в выражениях с булевыми операциями.

Символьные литералы

В Java символы представляют собой индексы в наборе символов Unicode. Это 16-битовые значения, которые могут быть преобразованы в целые значения и по отношению к которым можно выполнять целочисленные операции, такие как операции сложения и вычитания. Символьные литералы указываются внутри пары одинарных кавычек. Все отображаемые символы ASCII можно вводить непосредственно, указывая их в кавычках, например 'a', 'z' и '@'. Для ввода символов, непосредственный ввод которых невозможен, можно использовать несколько управляющих последовательностей, которые позволяют вводить нужные символы, такие как '\'' для символа одинарной кавычки и '\n' для символа новой строки. Существует также механизм для непосредственного ввода значения символа в восьмеричном или шестнадцатеричном виде. Для ввода значения в восьмеричной форме используют символ обратной косой черты, за которым следует трехзначный номер. Например, последовательность '\141' эквивалентна букве 'a'. Для ввода шестнадцатеричного значения применяются символы обратной косой черты и u (\u), за которыми следуют четыре шестнадцатеричные цифры. Например, '\u0061' представляет букву 'a' из набора символов ISO-Latin-1, поскольку старший байт является нулевым, а 'ua432' — это символ японской катаканы. Управляющие последовательности символов перечислены в табл. 3.3.

Таблица 3.3. Управляющие последовательности символов

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 (этот символ называют также символом перевода строки)

Окончание табл. 3.3

Управляющая последовательность	Описание
\f	Подача страницы
\t	Табуляция
\b	Возврат на одну позицию ("забой")

Строковые литералы

Указание строковых литералов в Java осуществляется так же, как в других языках, — заключаем последовательности символов в двойные кавычки. Вот несколько примеров строковых литералов.

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

Управляющие символы и восьмеричная/шестнадцатеричная формы записи, определенные для символьных литералов, работают точно так же и внутри строковых литералов. Важно отметить, что в Java строки должны начинаться и заканчиваться в одной строке. В этом языке отсутствует какой-либо управляющий символ продолжения строки, подобный тем, что имеются в ряде других языков.

На заметку! Как вы, возможно, знаете, в некоторых языках, включая C/C++, строки реализованы в виде массивов символов. Однако в Java это не так. В действительности строки представляют собой объектные типы. Как будет показано позднее, поскольку в Java строки реализованы в виде объектов, язык предлагает множество мощных и простых в использовании средств их обработки.

Переменные

Переменная — основной компонент хранения данных в программе Java. Переменная определяется комбинацией идентификатора, типа и необязательного начального значения. Кроме того, все переменные имеют область видимости, которая задает их видимость для других объектов, и продолжительность существования. Мы рассмотрим эти элементы в последующих разделах.

Объявление переменной

В Java все переменные должны быть объявлены до их использования. Основная форма объявления переменных выглядит следующим образом.

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

Здесь *тип* — это один из элементарных типов Java либо имя класса или интерфейса. (Типы класса и интерфейса рассмотрены в последующих главах этой части.) *идентификатор* — это имя переменной. Переменной можно присвоить начальное значение (инициализировать ее), указывая знак равенства и значение. Следует помнить, что выражение инициализации должно возвращать значение того же (или совместимого) типа, который указан для переменной. Для объявления нескольких переменных указанного типа можно использовать список, разделяемый запятыми.

Несколько примеров объявления переменных различных типов приведено ниже. Обратите внимание на то, что некоторые объявления осуществляют инициализацию переменных.

```
int a, b, c;          // объявление трех переменных типа int: a, b и c
int d = 3, e, f = 5; // объявление еще трех переменных типа int c
                      // инициализацией d и f
byte z = 22;          // инициализация переменной z
double pi = 3.14159; // объявление примерного значения переменной pi
char x = 'x';         // присваивание значения 'x' переменной x
```

Выбранные имена идентификаторов очень просты и указывают их тип. Язык Java допускает применение любого правильно оформленного идентификатора с любым объявленным типом.

Динамическая инициализация

Хотя в приведенных примерах в качестве начальных значений были использованы только константы, Java допускает динамическую инициализацию переменных с использованием любого выражения, допустимого в момент объявления переменной.

Например, ниже приведена короткая программа, которая вычисляет длину гипотенузы прямоугольного треугольника по длинам катетов.

```
// Этот пример демонстрирует динамическую инициализацию.
class DynInit {
    public static void main(String args[]) {
        double a = 3.0, b = 4.0;

        // динамическая инициализация переменной c
        double c = Math.sqrt(a * a + b * b);
        System.out.println("Гипотенуза равна " + c);
    }
}
```

Эта программа объявляет три локальные переменные — `a`, `b` и `c`. Две первые, `a` и `b`, инициализируются константами. Однако третья, `c`, инициализируется динамически, принимая значение длины гипотенузы (в соответствии с теоремой Пифагора). Для вычисления квадратного корня аргумента программа использует встроенный метод Java `sqrt()`, который является членом класса `Math`. В этом примере основной момент состоит в том, что в выражении инициализации можно использовать любые элементы, которые допустимы во время инициализации, в том числе вызовы методов, другие переменные или константы.

Область видимости и продолжительность существования переменных

До сих пор все использованные в примерах переменные были объявлены в начале метода `main()`. Однако Java допускает объявление переменных внутри любого блока. Как было сказано в главе 2, блок заключается в фигурные скобки. Он задает *область видимости*. Таким образом, при открытии каждого нового блока мы создаем новую область видимости. Область видимости задает то, какие объекты видимы другим частям программы. Она определяет также продолжительность существования этих объектов.

Многие другие языки программирования различают две основные категории области видимости: глобальную и локальную. Однако эти традиционные области

видимости не очень хорошо вписываются в строгую объектно-ориентированную модель Java. Хотя глобальную область видимости и можно задать, в настоящее время такой подход является скорее исключением, нежели правилом. В Java имеется две основные области видимости — определяемые классом и методом. Но даже это разделение несколько искусственно. Однако поскольку область видимости класса обладает несколькими уникальными свойствами и атрибутами, не применимыми к области видимости метода, такое разделение имеет определенный смысл. В связи с отличиями мы отложим рассмотрение области видимости класса (и объявленных внутри нее переменных) до главы 6, в которой описаны классы. А пока рассмотрим только те области видимости, которые определяются методом или внутри него.

Определенная методом область видимости начинается с его открывающей фигурной скобки. Однако если данный метод обладает параметрами, они также включаются в область видимости метода. Хота подробнее параметры рассмотрены в главе 6, пока отметим, что они работают точно так же, как любая другая переменная метода.

Основное правило, которое следует запомнить: переменные, объявленные внутри области видимости, не видны (т.е. недоступны) коду, который находится за пределами этой области. Таким образом, объявление переменной внутри области видимости ведет к ее локализации и защите от несанкционированного доступа и/или изменений. Действительно, правила обработки области видимости — основа инкапсуляции.

Области видимости могут быть вложенными. Например, при каждом создании блока кода мы создаем новую, вложенную, область видимости. В этих случаях внешняя область видимости заключает в себя внутреннюю область. Это означает, что объекты, объявленные во внешней области, будут видны коду, определенному во внутренней области. Тем не менее обратное не верно. Объекты, которые объявлены во внутренней области видимости, не будут видны за ее пределами.

Чтобы понять эффект применения вложенных областей видимости, рассмотрим следующую программу.

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

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

            // и x, и y известны в этой области видимости.
            System.out.println("x и y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100; // Ошибка! y не известна в этой области видимости

        // переменная x известна и здесь.
        System.out.println("x равна " + x);
    }
}
```

Как видно из комментариев, переменная x объявлена в начале области видимости метода `main()` и доступна всему последующему коду, находящемуся внутри этого метода. Объявление переменной y осуществляется внутри блока `if`. Поскольку блок задает область видимости, переменная y видна только коду внутри этого блока. Именно поэтому строка `y=100;`, расположенная вне этого блока, помещена в комментарий. Если удалить символ комментария, это приведет к ошибке времени компиляции, поскольку переменная y не видна за пределами своего блока. Переменную x можно ис-

пользовать внутри блока if, поскольку код внутри блока (т.е. во вложенной области видимости) имеет доступ к переменным, которые объявлены внешней областью.

Внутри блока переменные можно объявлять в любом месте, но они становятся допустимыми только после объявления. Таким образом, если переменная объявлена в начале метода, она доступна всему коду внутри этого метода. И наоборот, если переменная объявлена в конце блока, она, по сути, бесполезна, поскольку никакой код не получит к ней доступ. Например, следующий фрагмент кода неправилен, поскольку переменную count нельзя использовать до ее объявления.

```
// Этот фрагмент неправилен!
count = 100; // Стоп! Переменную count нельзя использовать до того,
             // как она будет объявлена!
int count;
```

Следует запомнить еще один важный нюанс: переменные создаются при входе в их область видимости и уничтожаются при выходе из нее. Это означает, что переменная утратит свое значение сразу по выходу из области видимости. Следовательно, переменные, которые объявлены внутри метода, не будут хранить свои значения между обращениями к этому методу. Кроме того, переменная, объявленная внутри блока, утратит свое значение по выходу из блока. Таким образом, продолжительность существования переменной ограничена ее областью видимости.

Если объявление переменной содержит инициализацию, инициализация переменной будет повторяться при каждом вхождении в блок, в котором она объявлена. Например, рассмотрим приведенную ниже программу.

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

        for(x = 0; x < 3; x++) {
            int y = -1; // y инициализируется при каждом вхождении
                        // в блок
            System.out.println("y равна: " + y); // эта строка всегда
                                                 // выводит значение -1
            y = 100;
            System.out.println("y теперь равна: " + y);
        }
    }
}
```

Эта программа создает следующий вывод.

```
y равна: -1
y теперь равна: 100
y равна: -1
y теперь равна: 100
y равна: -1
y теперь равна: 100
```

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

И последнее: хотя блоки могут быть вложенными, во внутреннем блоке нельзя объявлять переменные с тем же именем, что и во внешней области. Например, следующая программа ошибочна.

```
// Компиляция этой программы невозможна
class ScopeErr {
    public static void main(String args[]) {
```

```
        int bar = 1;
        {                   // создание новой области видимости
            int bar = 2; // Ошибка времени компиляции -
                            // переменная bar уже определена!
        }
    }
}
```

Преобразование и приведение типов

Те, кто уже обладает определенным опытом программирования, знают, что достаточно часто программисты присваивают переменной одного типа значение другого. Если оба типа совместимы, Java выполнит преобразование автоматически. Например, всегда можно присвоить значение типа `int` переменной типа `long`. Однако не все типы совместимы, и, следовательно, не все преобразования типов безоговорочно разрешены. Например, не существует никакого определенного автоматического преобразования типа `double` в тип `byte`. К счастью, преобразования между несовместимыми типами выполнять все-таки можно. Для этого необходимо использовать *приведение типов*, которое выполняет явное преобразование несовместимых типов. Рассмотрим автоматическое преобразование и приведение типов.

Автоматическое преобразование типов в Java

При присваивании данных переменной одного типа переменной другого типа выполняется *автоматическое преобразования типа* в случае удовлетворения следующих двух условий:

- оба типа совместимы;
- длина целевого типа больше длины исходного типа.

При соблюдении этих условий выполняется *преобразование с расширением*. Например, тип `int` всегда достаточно велик, чтобы хранить все допустимые значения типа `byte`, поэтому никакие операторы явного приведения типа не требуются.

С точки зрения преобразования с расширением, числовые типы, среди которых целочисленный и с плавающей точкой, совместимы друг с другом. Однако не существует автоматических преобразований числовых типов в тип `char` или `boolean`. Типы `char` и `boolean` также не совместимы и между собой.

Как уже говорилось ранее, Java выполняет автоматическое преобразование типов при сохранении целочисленной константы в переменных типа `byte`, `short`, `long` или `char`.

Приведение несовместимых типов

Хотя автоматическое преобразование типов удобно, оно не в состоянии удовлетворить все потребности. Например, что делать, если нужно присвоить значение типа `int` переменной типа `byte`? Это преобразование не будет выполняться автоматически, поскольку тип `byte` меньше типа `int`. Иногда этот вид преобразования называют *преобразованием с сужением*, поскольку значение явно сужается, чтобы оно могло уместиться в целевом типе.

Чтобы выполнить преобразование между двумя несовместимыми типами, необходимо использовать приведение типов. *Приведение* – это всего лишь явное преобразование типов. Общая форма преобразования имеет вид.

```
(целевой_тип) значение
```

Здесь *целевой_тип* определяет тип, в который нужно преобразовать указанное значение. Например, следующий фрагмент кода приводит тип `int` к типу `byte`. Если значение целочисленного типа больше допустимого диапазона типа `byte`, оно будет уменьшено до результата деления по модулю (остатка от целочисленного деления) на диапазон типа `byte`.

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

При присваивании значения с плавающей точкой переменной целочисленного типа выполняется другой вид преобразования типа – *усечение*. Как вы знаете, целые числа не содержат дробной части. Таким образом, когда значение с плавающей точкой присваивается переменной целочисленного типа, дробная часть отбрасывается. Например, в случае присваивания значения 1,23 целочисленной переменной результирующим значением будет просто 1. Дробная часть – 0,23 – усекается. Конечно, если размер целочисленной части слишком велик, чтобы уместиться в целевом целочисленном типе, значение будет уменьшено до результата деления по модулю на диапазон целевого типа.

Следующая программа демонстрирует ряд преобразований типа, которые требуют приведения.

```
// Демонстрация приведения типов.
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);
    }
}
```

Эта программа создает следующий вывод.

```
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
```

Рассмотрим каждое из этих преобразований. Когда значение 257 приводится к типу byte, результатом будет остаток от деления 257 на 256 (диапазон допустимых значений типа byte), который в данном случае равен 1. Когда значение переменной d преобразуется в тип int, его дробная часть отбрасывается. Когда значение переменной d преобразуется в тип byte, его дробная часть отбрасывается и значение уменьшается до результата деления по модулю на 256, который в данном случае равен 67.

Автоматическое повышение типа в выражениях

Кроме операций присваивания, определенное преобразование типов может выполняться также в выражениях. Для примера рассмотрим следующую ситуацию. Иногда в выражениях для обеспечения необходимой точности промежуточное значение может выходить за пределы допустимого диапазона любого из операндов. Например, рассмотрим следующее выражение.

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

Результат вычисления промежуточного члена a*b вполне может выйти за пределы диапазона допустимых значений его операндов типа byte. Для решения подобных проблем при вычислении выражений Java автоматически повышает тип каждого операнда byte или short до int. То есть вычисление промежуточного выражения a*b выполняется с применением целочисленных значений, а не байтов. Поэтому результат промежуточного выражения 50*40, равный 2000, оказывается допустимым, несмотря на то что и для a, и для b задан тип byte.

Хотя автоматическое повышение типа очень удобно, оно может приводить к досадным ошибкам во время компиляции. Например, следующий внешне абсолютно корректный код приводит к возникновению проблемы.

```
byte b = 50;
b = b * 2; // Ошибка! Значение типа int не может быть присвоено
           // переменной типа byte!
```

Код предпринимает попытку повторного сохранения произведения 50*2 — совершенно допустимого значения типа byte — в переменной типа byte. Однако поскольку во время вычисления выражения тип операндов был автоматически повышен до int, тип результата также был повышен до int. Таким образом, теперь результат выражения имеет тип int, который не может быть присвоен переменной типа byte без приведения типа. Сказанное справедливо даже тогда, когда, как в данном конкретном случае, значение, которое должно быть присвоено, умещается в переменной целевого типа.

В тех случаях, когда последствия переполнения понятны, следует использовать явное приведение типов

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

которое приводит к правильному значению, равному 100.

Правила повышения типа

В Java определено несколько правил *повышения типа*, применяемых к выражениям. Во-первых, тип всех значений `byte`, `short` и `char` повышается до `int`, как было описано в предыдущем разделе. Во-вторых, если один операнд имеет тип `long`, тип всего выражения повышается до `long`. Если один операнд имеет тип `float`, тип всего выражения повышается до `float`. Если любой из операндов имеет тип `double`, типом результата будет `double`.

Следующая программа демонстрирует повышение типа значения одного из операндов к типу второго в каждом операторе с двумя операндами:

```
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 подробнее рассмотрим повышение типа, выполняемое в следующей строке программы.

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

В первом промежуточном выражении, `f*b`, тип переменной `b` повышается до `float`, и типом результата вычисления этого промежуточного выражения также является `float`. В следующем промежуточном выражении `i/c` тип `c` повышается до `int`, и результат вычисления этого выражения — `int`. Затем в выражении `d*s` тип значения `s` повышается до `double`, и все промежуточное выражение получает тип `double`. И наконец, выполняются операции с этими тремя промежуточными значениями типов `float`, `int` и `double`. Результат сложения значений типов `float` и `int` имеет тип `float`. Затем тип значения разности результирующего значения типа `float` и последнего значения типа `double` повышается до `double`, который и становится окончательным типом результата выражения.

Массивы

Массив — это группа однотипных переменных, для обращения к которым используется общее имя. Java допускает создание массивов любого типа, которые могут иметь одно или несколько измерений. Доступ к конкретному элементу массива осуществляется по его индексу. Массивы предлагают удобный способ группирования связанной информации.

На заметку! Те, кто знаком с языками C/C++, должны быть особенно внимательны. В Java массивы работают не так, как в этих языках.

Одномерные массивы

По сути, *одномерные массивы* представляют собой список однотипных переменных. Чтобы создать массив, вначале необходимо создать переменную массива тре-

буемого типа. Общая форма объявления одномерного массива выглядит следующим образом.

```
тип имя_переменной[];
```

Здесь *тип* задает тип элемента (называемый также базовым типом) массива. Тип элемента определяет тип данных каждого из элементов, составляющих массив. Таким образом, тип элемента массива определяет тип данных, которые будет содержать массив. Например, следующий оператор объявляет массив month_days, имеющий тип "массив элементов типа int".

```
int month_days[];
```

Хотя это объявление утверждает, что month_days — массив переменных, в действительности никакого массива еще не существует. Фактически значение массива month_days установлено равным null, которое представляет массив без значений. Чтобы связать имя month_days с реальным физическим массивом целочисленных значений, необходимо с помощью оператора new зарезервировать память и присвоить ее массиву month_days.

Подробнее мы рассмотрим эту операцию в следующей главе, однако она нужна сейчас для выделения памяти под массивы. Общая форма оператора new применительно к одномерным массивам выглядит следующим образом.

```
переменная_массива = new тип [размер];
```

Здесь *тип* определяет тип данных, для которых резервируется память, *размер* указывает количество элементов в массиве, а *переменная_массива* — переменная массива, связанная с массивом. Другими словами, чтобы использовать оператор new для резервирования памяти, потребуется указать тип и количество элементов, для которых нужно зарезервировать память. Элементы массива, для которых память была выделена оператором new, будут автоматически инициализированы нулевыми значениями (для числовых типов), значениями false (для логического типа) или значениями null (для ссылочных типов, рассматриваемых в следующей главе). Приведенный ниже оператор резервирует память для 12-элементного массива целых значений и связывает их с массивом month_days.

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

После выполнения этого оператора month_days будет ссылаться на массив, состоящий из 12 целочисленных значений. При этом все элементы массива будут инициализированы нулевыми значениями.

Подведем итоги: создание массива — двухэтапный процесс. Во-первых, необходимо объявить переменную нужного типа массива. Во-вторых, с помощью оператора new необходимо зарезервировать память для хранения массива и присвоить ее переменной массива. Таким образом, в Java все массивы являются динамически распределяемыми. Если вы еще не знакомы с концепцией динамического распределения памяти, не беспокойтесь. Этот процесс будет подробно описан в последующих главах книги.

Как только массив создан и память для него зарезервирована, к конкретному элементу массива можно обращаться, указывая его индекс в квадратных скобках. Индексы массива начинаются с нуля. Например, следующий оператор присваивает значение 28 второму элементу массива month_days.

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

Следующая строка кода отображает значение, хранящееся в элементе с индексом, равным 3.

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

Чтобы продемонстрировать весь процесс в целом, приведем программу, которая создает массив количества 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.");
    }
}
```

Если выполнить эту программу, она выведет количество дней в апреле. Как уже было сказано, в Java индексация элементов массивов начинается с нуля, поэтому количество дней в апреле — `month_days[3]`, или 30.

Объявление переменной массива можно объединять с распределением самого массива, как показано в следующем примере.

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

Именно так обычно и поступают в профессионально написанных программах Java. Массивы можно инициализировать при их объявлении. Этот процесс во многом аналогичен инициализации простых типов. *Инициализатор массива* — это разделяемый запятыми список выражений, заключенный в фигурные скобки. Запятые разделяют значения элементов массива. Массив будет автоматически создан достаточно большим, чтобы в нем могли уместиться все элементы, указанные в инициализаторе массива. В этом случае использование оператора `new` не требуется. Например, чтобы сохранить количество дней каждого месяца, можно использовать следующий код, который создает и инициализирует массив целых значений.

```
// Усовершенствованная 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.");
    }
}
```

При выполнении эта программа выдает такой же результат, как и предыдущая версия.

Система Java выполняет тщательную проверку, чтобы убедиться в том, не была ли случайно предпринята попытка присвоения или обращения к значениям, которые выходят за пределы допустимого диапазона массива. Система времени выполнения Java будет проверять соответствие всех индексов массива допустимому диапазону. Например, система времени выполнения будет проверять соответствие значения каждого индекса допустимому диапазону от 0 до 11 включительно. Попытка обращения к элементам за пределами диапазона массива (указание отри-

цательных индексов или индексов, которые превышают длину массива) приведет к ошибке времени выполнения.

Приведем еще один пример программы, в которой используется одномерный массив. Эта программа вычисляет среднее значение набора чисел.

```
// Вычисление среднего значения 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 равно " + result / 5);
    }
}
```

Многомерные массивы

В Java *многомерные массивы* представляют собой массивы массивов. Они, как можно догадаться, выглядят и действуют подобно обычным многомерным массивам. Однако, как вы увидите, они имеют несколько незначительных отличий. При объявлении переменной многомерного массива для указания каждого дополнительного индекса используют отдельный набор квадратных скобок. Например, следующий код объявляет переменную двухмерного массива `twoD`.

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

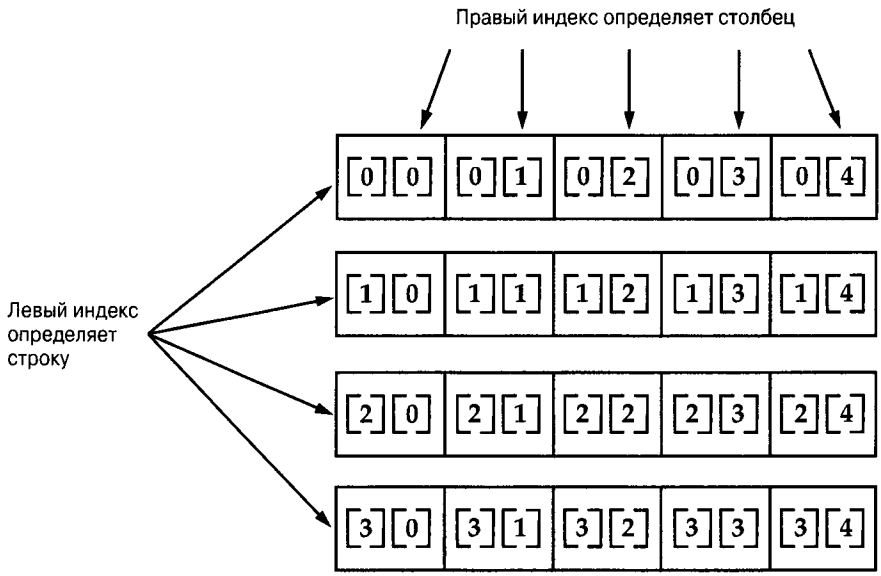
Этот оператор резервирует память для массива размерностью 4×5 и присваивает его переменной `twoD`. Внутренне эта матрица реализована как *массив массивов* значений типа `int`. С точки зрения логической организации этот массив будет выглядеть подобно изображенному на рис. 3.1.

Следующая программа нумерует элементы в массиве слева направо, сверху вниз, а затем отображает эти значения.

```
// Демонстрация двухмерного 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();
        }
    }
}
```



Дано: `int twoD[][] = new int[4][5];`

Рис. 3.1. Логическое представление двухмерного массива 4×5

Эта программа создает следующий вывод.

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

При резервировании памяти под многомерный массив необходимо указать память только для первого (левого) измерения. Для каждого из остальных измерений память можно резервировать отдельно. Например, следующий код резервирует память для первого измерения массива `twoD` при его объявлении. Резервирование памяти для второго измерения массива осуществляется вручную.

```
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 данной ситуации отдельное резервирование памяти для второго измерения массива и не дает никаких преимуществ, в других ситуациях это может быть полезно. Например, при резервировании памяти для измерений массива вручную, не нужно резервировать одинаковое количество элементов для каждого измерения. Как было отмечено ранее, поскольку в действительности многомерные массивы представляют собой массивы массивов, программист полностью управляет длиной каждого массива. Например, следующая программа создает двухмерный массив с различными размерами второго измерения.

```
// Резервирование памяти вручную для массива с различными
// размерами второго измерения.
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();
        }
    }
}
```

Эта программа создает следующий вывод.

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

Созданный ею массив выглядит, как показано на рис. 3.2.

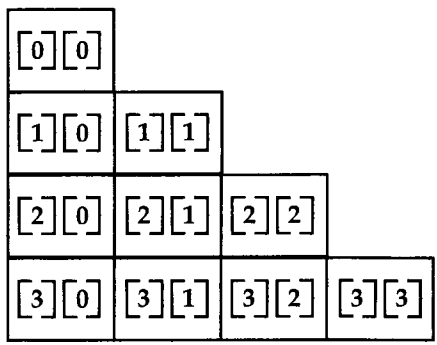


Рис. 3.2. Двухмерный массив с различными размерами второго измерения

Использование неоднородных (или нерегулярных) массивов может быть не применимо во многих приложениях, поскольку их поведение отличается от обычного поведения многомерных массивов. Однако в некоторых ситуациях нерегулярные массивы могут оказаться весьма эффективными. Например, нерегулярный массив может быть идеальным решением, если требуется очень большой двухмерный разреженный массив (т.е. массив, в котором будут использоваться не все элементы).

Многомерные массивы можно инициализировать. Для этого достаточно заключить инициализатор каждого измерения в отдельный набор фигурных скобок. Следующая программа создает матрицу, в которой каждый элемент содержит произведение индексов строки и столбца. Обратите также внимание на то, что внутри инициализаторов массивов можно применять как литеральное значение, так и выражения.

```
// Инициализация двухмерного 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();
        }
    }
}
```

При выполнении эта программа создает следующий 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, каждая строка массива инициализируется в соответствии со значениями, указанными в списках инициализации.

Рассмотрим еще один пример использования многомерного массива. Следующая программа создает трехмерный массив размерностью 3×4×5. Затем она загружает каждый элемент произведением его индексов и, наконец, отображает эти произведения.

```
// Демонстрация 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();
        }
    }
}
```

Эта программа создает следующий вывод.

```
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

Для объявления массивов можно использовать и вторую форму.

тип`[]` *имя_переменной*`;`

В этой форме квадратные скобки следуют за указателем типа, а не за именем переменной массива. Например, следующие два объявления эквивалентны.

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

Приведенные ниже два объявления также эквивалентны.

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

Вторая форма объявления удобна для одновременного объявления нескольких массивов. Например, объявление

```
int[] nums, nums2, nums3; // создание трех массивов
```

создает три переменные массивов типа `int`. Оно эквивалентно следующему объявлению.

```
int nums[], nums2[], nums3[]; // создание трех массивов
```

Альтернативная форма объявления удобна также при указании массива в качестве возвращаемого типа метода. В этой книге используются обе формы объявлений.

Несколько слов о строках

Как вы, вероятно, заметили, в ходе рассмотрения типов данных и массивов мы не упоминали строки или строковый тип данных. Это связано не с тем, что язык Java не поддерживает этот тип, — он его поддерживает. Просто строковый тип данных Java, имеющий имя `String`, не относится к элементарным типам. Он не является также и просто массивом символов. Строка, скорее, представляет собой объект, и для понимания его полного описания требуется понимание ряда характеристик объектов. Поэтому рассмотрим этот тип в последующих главах книги, после рассмотрения объектов. Однако чтобы читатели могли использовать простые строки в примерах программ, мы сейчас кратко опишем этот тип.

Тип `String` используют для объявления строковых переменных. Можно также объявлять массивы строк. Переменной типа `String` можно присваивать заключенную в кавычки строковую константу. Переменная типа `String` может быть присвоена другой переменной типа `String`. Объект класса `String` можно применять в качестве аргумента метода `println()`. Например, рассмотрим следующий фрагмент кода.

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

В этом примере `str` — объект класса `String`. Ему присвоена строка `"тестовая строка"`, которая отображается методом `println()`.

Как будет показано в дальнейшем, объекты класса `String` обладают многими характерными особенностями и атрибутами, которые делают их достаточно мощными и простыми в использовании. Однако в нескольких последующих главах мы будем применять их только в простейшей форме.

Замечание по поводу указателей для программистов на C/C++

Опытные программисты на C/C++ знают, что эти языки поддерживают указатели. Однако в настоящей главе мы о них не упоминали. Причина этого проста: Java не поддерживает и не разрешает использование указателей. (Точнее говоря, Java не поддерживает указатели, которые доступны и/или могут быть изменены программистом.) Язык Java не разрешает использование указателей, поскольку это позволило бы программам Java преодолевать защитный барьер между средой исполнения Java и содержащим ее компьютером. (Вспомните, что указателю может быть присвоен любой адрес в памяти — даже те адреса, которые могут находиться вне системы времени выполнения Java.) Поскольку в программах C/C++ указатели используются достаточно интенсивно, их утрата может казаться существенным недостатком Java. В действительности это не так. Среда Java спроектирована так, чтобы до тех пор, пока все действия выполняются в пределах среды исполнения, применение указателей не требовалось, и их использование не дает никаких преимуществ.

ГЛАВА 4

Операторы

Язык Java предоставляет множество операторов. Большинство из них может быть отнесено к одной из следующих четырех групп: арифметические операторы, побитовые операторы, операторы сравнения и логические операторы. В Java также определен ряд дополнительных операторов, применяемых в особых ситуациях. В этой главе описаны все операторы Java, за исключением оператора сравнения типов `instanceof`, который рассматривается в главе 13.

Арифметические операторы

Арифметические операторы используются в математических выражениях, так как они применяются в алгебре. Арифметические операторы перечислены в табл. 4.1.

Таблица 4.1. Арифметические операторы Java

Оператор	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 операторов должны иметь числовой тип. Арифметические операторы нельзя применять к логическим типам, но можно применять к типам `char`, поскольку в Java этот тип, по сути, является разновидностью типа `int`.

Основные арифметические операторы

Все основные арифметические операторы — сложение, вычитание, умножение и деление — воздействуют на числовые типы так, как этого можно было бы ожидать. Оператор унарного вычитания изменяет знак своего единственного операнда. Оператор унарной суммы просто возвращает значение своего операнда. Следует помнить, что в случае применения оператора деления к целочисленному типу результат не будет содержать дробного компонента.

Следующий пример простой программы демонстрирует применение арифметических операторов. Он иллюстрирует также различие между делением с плавающей точкой и целочисленным делением.

```
// Демонстрация основных арифметических операторов.
class BasicMath {
    public static void main(String args[]) {
        // арифметические операции с целочисленными значениями
        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 типа 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);
    }
}
```

При выполнении этой программы на экране отобразится следующий 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
```

Оператор деления по модулю

Оператор деления по модулю, %, возвращает остаток деления. Этот оператор можно применять как к типам с плавающей точкой, так и к целочисленным типам. Следующий пример программы демонстрирует применение оператора %.

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

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

При выполнении эта программа создает следующий вывод.

```
x mod 10 = 2
y mod 10 = 2.25
```

Составные арифметические операторы с присваиванием

В языке Java имеются специальные операторы, объединяющие арифметические операторы с операцией присваивания. Как вы, вероятно, знаете, операторы вроде показанного ниже в программах встречаются достаточно часто.

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

В Java этот оператор можно записать следующим образом.

```
a += 4;
```

В этой версии использован *составной оператор присваивания* +=. Оба оператора выполняют то же действие: они увеличивают значение переменной a на 4. А вот еще один пример.

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

Этот пример можно записать следующим образом.

```
a %= 2;
```

В этом случае оператор %= вычисляет остаток от деления a/2 и помещает результат обратно в переменную a. Составные операторы с присваиванием существуют для всех арифметических операторов с двумя операндами. Таким образом, любой оператор, имеющий форму

переменная = переменная оператор выражение;

можно записать в следующем виде.

переменная оператор= выражение;

Составные операторы с присваиванием предоставляют два преимущества. Во-первых, они позволяют уменьшить объем вводимого кода, поскольку являются "сокращенным" вариантом соответствующих длинных форм. Во-вторых, их реализация в системе времени выполнения Java эффективнее реализации эквивалентных длинных форм. Поэтому в профессионально написанных программах Java составные операторы с присваиванием будут встречаться очень часто.

Ниже приведен пример программы, демонстрирующий практическое применение нескольких составных операторов с присваиванием.

```
// Демонстрация применения нескольких операторов с присваиванием.
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);
    }
}
```

Эта программа создает следующий вывод.

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

Инкремент и декремент

Операторы ++ и -- являются операторами инкремента и декремента. Они были представлены в главе 2. А в этой главе мы рассмотрим их подробно. Как вы вскоре убедитесь, эти операторы имеют ряд особых свойств, которые делают их достаточно интересными. Рассмотрим, что именно делают операторы инкремента и декремента.

Оператор инкремента увеличивает значение операнда на единицу. Оператор декремента уменьшает значение операнда на единицу. Например, выражение

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

с применением оператора инкремента можно записать в таком виде.

```
x++;
```

Аналогично выражение

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

эквивалентно следующему выражению.

```
x--;
```

Эти операторы отличаются тем, что могут быть записаны как в *постфиксной* форме, когда оператор следует за операндом, как в приведенных примерах, так и в *префиксной*, когда он предшествует операнду. В приведенных примерах применение любой из этих форм не имеет никакого значения. Однако, когда операторы инкремента и декремента являются частью более сложного выражения, проявляется внешне незначительное, но важное различие между этими двумя формами. В префиксной форме значение операнда увеличивается или уменьшается до извлечения значения для использования в выражении. В постфиксной форме предыдущее значение извлекается для использования в выражении, и лишь после этого значение операнда изменяется.

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

В этом случае значение y устанавливается равным 43, как и можно было ожидать, поскольку увеличение значения выполняется перед присваиванием значения переменной x переменной y. Таким образом, строка y=++x эквивалентна следующим двум операторам.

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

Однако если операторы записать как

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

значение переменной x извлекается до выполнения оператора инкремента, и поэтому значение переменной y равно 42. Конечно, в обоих случаях значение переменной x установлено равным 43. Следовательно, строка y=x++; эквивалентна следующим двум операторам.

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

Следующая программа демонстрирует применение оператора инкремента.

```
// Демонстрация применения оператора ++.
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);
    }
}
```

Вывод этой программы выглядит следующим образом.

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

Побитовые операторы

Язык Java определяет несколько *побитовых операторов*, которые могут применяться к целочисленным типам: long, int, short, char и byte. Эти операторы воздействуют на отдельные биты операндов. Они перечислены в табл. 4.2.

Поскольку побитовые операторы манипулируют битами в целочисленном значении, важно понимать, какое влияние подобные манипуляции могут оказать на значение. В частности, важно знать, как среда Java хранит целочисленные значения и как она представляет отрицательные числа. Поэтому, прежде чем продолжить рассмотрение операторов, кратко рассмотрим эти два вопроса.

Таблица 4.2. Побитовые операторы в Java

Оператор	Описание
~	Побитовый унарный оператор NOT (НЕ)
&	Побитовое AND (И)
|	Побитовое OR (ИЛИ)
^	Побитовое исключающее OR
>>	Сдвиг вправо
>>>	Сдвиг вправо с заполнением нулями
<<	Сдвиг влево
&=	Побитовое AND с присваиванием
|=	Побитовое OR с присваиванием
^=	Побитовое исключающее OR с присваиванием
>>=	Сдвиг вправо с присваиванием
>>>=	Сдвиг вправо с заполнением нулями с присваиванием
<<=	Сдвиг влево с присваиванием

Все целочисленные типы представляются двоичными числами различной длины. Например, значение типа `byte`, равное 42, в двоичном представлении имеет вид 00101010, в котором каждая позиция представляет степень числа два, начиная с 20 в крайнем справа бите. Бит в следующей позиции будет представлять 21, или 2, следующий – 22, или 4, затем 8, 16, 32 и т.д. Таким образом, двоичное представление числа 42 содержит единичные биты в позициях 1, 3 и 5 (начиная с 0, крайней справа позиции). Следовательно, 42=21+23+25=2+8+32.

Все целочисленные типы (за исключением `char`) — целочисленные типы со знаком. Это означает, что они могут представлять как положительные, так и отрицательные значения. В Java применяется кодирование, называемое двоичным дополнением, при котором отрицательные числа представляются в результате инвертирования всех битов значения (изменения 1 на 0 и наоборот) и последующего добавления 1 к результату. Например, –42 представляется в результате инвертирования всех битов в двоичном представлении числа 42, что дает значение 11010101, и добавления 1, что приводит к значению 11010110, или –42. Чтобы декодировать отрицательное число, необходимо вначале инвертировать все биты, а затем добавить 1 к результату. Например, инвертирование значения –42, или 11010110, приводит к значению 00101001, или 41, после добавления 1 к которому мы получаем 42.

Причина, по которой в языке Java (и большинстве других компьютерных языков) применяют двоичное дополнение, становится понятной при рассмотрении перехода через нуль. Если речь идет о значении типа `byte`, нуль представляется значением 00000000. В случае применения единичного дополнения простое инвертирование всех битов создает значение, равное 11111111, которое представляет отрицательный нуль. Проблема в том, что отрицательный нуль — недопустимое значение в целочисленной математике. Применение двоичного дополнения для представления отрицательных значений позволяет решить эту проблему. При этом к дополнению добавляется 1, что приводит к числу 100000000. Единичный бит оказывается сдвинутым влево слишком далеко, чтобы умещаться в значении типа `byte`. Тем самым достигается требуемое поведение, когда -0 эквивалентен 0, а 11111111 — код значения, равного -1. Хотя в приведенном примере мы использовали значение типа `byte`, тот же базовый принцип применяется и ко всем целочисленным типам Java.

Поскольку в Java для хранения отрицательных значений используется двоичное дополнение — и поскольку в Java все целочисленные значения являются значениями со знаком — применение побитовых операторов может легко привести к неожиданным результатам. Например, установка самого старшего бита равным 1 может привести к тому, что результирующее значение будет интерпретироваться как отрицательное число, независимо от того, к этому результату вы стремились или нет. Во избежание неприятных сюрпризов следует помнить, что независимо от того, как он был установлен, старший бит определяет знак целого числа.

Побитовые логические операторы

Побитовые логические операторы — это &, |, ^ и ~. Результаты выполнения каждого из этих операторов приведены в табл. 4.3. В ходе ознакомления с последующим материалом помните, что побитовые операторы применяются к каждому отдельному биту каждого операнда.

Таблица 4.3. Результаты выполнения побитовых логических операторов

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 NOT

Унарный оператор NOT (НЕ), ~, называемый также *побитовым дополнением*, инвертирует все биты операнда. Например, число 42, которое имеет следующую последовательность битов:

```
00101010
```

в результате применения оператора NOT преобразуется следующим образом.

```
11010101
```

Побитовое AND

Значение бита, полученное в результате выполнения побитового оператора AND, &, равно 1, если соответствующие биты в операндах также равны 1. Во всех остальных случаях значение результирующего бита равно 0.

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

Побитовое OR

Результирующий бит, полученный в результате выполнения оператора OR, |, равен 1, если соответствующий бит в любом из операндов равен 1, как показано в следующем примере.

```
  00101010    42
| 00001111    15
```

```
----------------
  00101111    47
```

Побитовое XOR

Результирующий бит, полученный в результате выполнения оператора XOR, ^, равен 1, если соответствующий бит только в одном из операндов равен 1. Во всех других случаях результирующий бит равен 0. В следующем примере показано применение оператора ^. Он демонстрирует также полезную особенность оператора XOR. Обратите внимание на инвертирование последовательности битов числа 42 во всех случаях, когда второй операнд содержит бит, равный 1. Во всех случаях, когда второй операнд содержит бит, равный 0, значение первого операнда остается неизменным. Это свойство пригодится при выполнении некоторых операций с битами.

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

Использование побитовых логических операторов

В следующей программе демонстрируется применение побитовых логических операторов.

```
// Демонстрация побитовых логических операторов.
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 двоичном представлении
        int b = 6; // 4 + 2 + 0 или 0110 в двоичном представлении
        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 последовательности битов переменных a и b представляют все четыре возможные комбинации двух двоичных цифр: 0-0, 0-1, 1-0 и 1-1. О действии операторов | и & на каждый бит можно судить по результирующим значениям переменных c и d. Значения, присвоенные переменным e и f, иллюстрируют действие оператора ^. Массив строк binary содержит читабельные двоичные представления чисел от 0 до 15. В этом примере массив индексирован, что позволяет увидеть двоичное представление каждого результирующего значения. Массив построен так, чтобы соответствующее строковое представление двоичного значения n хранилось в элементе массива binary[n]. Чтобы его можно было вывести при помощи массива binary, значение ~a уменьшается до значения, меньшего 16,

в результате его объединения со значением 0x0f (0000 1111 в двоичном представлении) оператором AND. Вывод этой программы имеет следующий вид.

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

Сдвиг влево

Оператор сдвига влево, <<, смещает все биты значения влево на указанное количество позиций. Он имеет следующую общую форму.

значение << *количество*

Здесь *количество* — количество позиций, на которое нужно сдвинуть влево биты в значении *значение*. То есть оператор << смещает влево биты указанного значения на количество позиций, указанных операндом *количество*. При каждом сдвиге влево самый старший бит смещается за пределы допустимого диапазона (и теряется), а справа дописывается нуль. Это означает, что при применении оператора сдвига влево к операнду типа `int` биты теряются, как только они сдвигаются за пределы 31 позиции. Если операнд имеет тип `long`, биты теряются после сдвига за пределы 63 позиции.

Автоматическое повышение типа, выполняемое в среде Java, приводит к непредвиденным результатам при выполнении сдвига в значениях типа `byte` и `short`. Как вы уже знаете, тип значений `byte` и `short` повышается до типа `int` при вычислении выражений. Более того, результат вычисления такого выражения также имеет тип `int`. Это означает, что результатом выполнения сдвига влево значения типа `byte` или `short` будет значение типа `int`, и сдвинутые влево биты не будут отброшены до тех пор, пока они не будут сдвинуты за пределы 31 позиции. Более того, при повышении до типа `int` отрицательное значение типа `byte` или `short` получит дополнительный знаковый разряд. Следовательно, старшие биты будут заполнены единицами. Поэтому выполнение оператора сдвига влево применительно к значению типа `byte` или `short` предполагает необходимость отбрасывания старших байтов результата типа `int`. Например, при выполнении сдвига влево в значении типа `byte` вначале будет осуществляться повышение типа значения до типа `int` и лишь затем сдвиг. Это означает, что для получения требуемого сдвинутого значения типа `byte` необходимо отбросить три старших байта результата. Простейший способ достижения этого — обратное приведение результата к типу `byte`. Следующая программа демонстрирует эту концепцию.

```
// Сдвиг влево значения типа 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 and b: " + i + " " + b);
    }
}
```

Эта программа создает следующий вывод.

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

Поскольку для выполнения вычислений тип переменной a повышается до int, сдвиг влево на две позиции значения 64 (0100 0000) приводит к значению i, равному 256 (1 0000 0000). Однако переменная b содержит значение, равное 0, поскольку после сдвига младший байт равен 0. Единственный единичный бит оказывается сдвинутым за пределы допустимого диапазона.

Поскольку каждый сдвиг влево на одну позицию, по сути, удваивает исходное значение, программисты часто используют это в качестве эффективной замены умножения на 2. Однако при этом следует соблюдать осторожность. При сдвиге единичного бита в старшую позицию (бит 31 или 63) значение становится отрицательным. Следующая программа демонстрирует это применение оператора сдвига влево.

```
// Применение сдвига влево в качестве быстрого метода 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);
        }
    }
}
```

Программа создает следующий вывод.

```
536870908
1073741816
2147483632
-32
```

Начальное значение было специально выбрано таким, чтобы после сдвига влево на 4 позиции оно стало равным -32. Как видите, после сдвига единичного бита в позицию 31 число интерпретируется как отрицательное.

Сдвиг вправо

Оператор сдвига вправо, >>, смещает все биты значения вправо на указанное количество позиций. В общем виде его можно записать следующим образом.

```
значение >> количество
```

Здесь *количество* указывает количество позиций, на которое нужно сдвинуть вправо биты в значении *значение*. То есть оператор >> перемещает все биты в указанном значении вправо на количество позиций, указанное операндом *количество*.

Следующий фрагмент кода выполняет сдвиг вправо на две позиции в значении 32, в результате чего значение переменной a становится равным 8.

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

Когда какие-либо биты в значении "сдвигаются прочь", они теряются. Например, следующий фрагмент кода выполняет сдвиг вправо на две позиции в значении 35, что приводит к потере двух младших битов и повторной установке значения переменной a равным 8.

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

Чтобы лучше понять, как выполняется этот оператор, рассмотрим его применение к двоичным представлениям.

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

При каждом сдвиге вправо выполняется деление значения на два с отбрасыванием любого остатка. Это свойство можно использовать для высокопроизводительного целочисленного деления на 2. Конечно, при этом нужно быть уверенным, что никакие биты не будут сдвинуты за пределы правой границы.

При выполнении сдвига вправо старшие (расположенные в крайних левых позициях) биты, освобожденные в результате сдвига, заполняются предыдущим содержимым старшего бита. Этот эффект называется *дополнительным знаковым разрядом* и служит для сохранения знака отрицательных чисел при их сдвиге вправо. Например, результат выполнения оператора `-8>>1` равен –4, что в двоичном представлении выглядит следующим образом.

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

Интересно отметить, что результат сдвига вправо значения –1 всегда равен –1, поскольку дополнительные знаковые разряды добавляют новые единицы к старшим битам.

Иногда при выполнении сдвига вправо появление дополнительных знаковых разрядов нежелательно. Например, следующая программа преобразует значение `byte` в соответствующее шестнадцатеричное строковое представление. Обратите внимание на то, что для обеспечения возможности использования значения в качестве индекса массива шестнадцатеричных символов сдвинутое значение маскируется за счет его объединения со значением `0x0f` оператором AND, что приводит к отбрасыванию любых битов дополнительных знаковых разрядов.

```
// Маскирование дополнительных знаковых разрядов.
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]);
    }
}
```

Вывод этой программы выглядит следующим образом.

```
b = 0xf1
```

Сдвиг вправо без учета знака

Как было показано, при каждом выполнении оператор >> автоматически заполняет старший бит его предыдущим содержимым. В результате знак значения сохраняется. Однако иногда это нежелательно. Например, при выполнении сдвига вправо в каком-либо значении, которое не является числовым, использование дополнительных знаковых разрядов может быть нежелательным. Эта ситуация часто встречается при работе со значениями пикселей и графическими изображениями. Как правило, в этих случаях требуется сдвиг нуля в позицию старшего бита независимо от его первоначального значения. Такое действие называют *сдвигом вправо без учета знака*. Для его выполнения используют оператор сдвига вправо без учета знака Java, >>>, который всегда вставляет нуль в позицию старшего бита.

Следующий фрагмент кода демонстрирует применение оператора >>>. В этом примере значение переменной a установлено равным –1, все 32 бит двоичного представления которого равны 1. Затем в этом значении выполняется сдвиг вправо на 24 бит с заполнением старших 24 бит нулями и игнорированием обычно используемых дополнительных знаковых разрядов. В результате значение a становится равным 255.

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

Чтобы происходящее было понятнее, запишем эту же операцию в двоичной форме.

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

Часто оператор >>> не столь полезен, как хотелось бы, поскольку он имеет смысл только для 32- и 64-разрядных значений. Помните, что в выражениях тип меньших значений автоматически повышается до int. Это означает применение дополнительных знаковых разрядов и выполнение сдвига по отношению к 32-разрядным, а не 8- или 16-разрядным значениям. То есть программист может подразумевать выполнение сдвига вправо без учета знака применительно к значению типа byte и заполнение нулями начиная с бита 7.

Однако в действительности это не так, поскольку фактически сдвиг будет выполняться в 32-разрядном значении. Этот эффект демонстрирует следующая программа.

```
// Сдвиг без учета знака 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]);
    }
}
```

Из следующего вывода этой программы видно, что оператор >>> не выполняет никаких действий по отношению к значениям типа byte. Для этого примера в качестве значения переменной b было выбрано произвольное отрицательное значение типа byte. Затем переменной c присваивается значение переменной b типа byte, смещенное вправо на четыре позиции, которое в связи с применением дополнительных знаковых разрядов равно 0xff. Затем переменной d присваивается значение переменной b типа byte, сдвинутое вправо на четыре позиции без учета знака, которым должно было бы быть значение 0x0f, но в действительности, из-за применения дополнительных знаковых разрядов во время повышения типа переменной b до int перед выполнением сдвига, значением оказывается 0xff. Последнее выражение устанавливает значение переменной e равным значению типа byte переменной b, замаскированному до 8 бит с помощью оператора AND, а затем сдвинутому вправо на четыре позиции, что дает ожидаемое значение, равное 0x0f. Обратите внимание на то, что оператор сдвига вправо без учета знака не применялся к переменной d, поскольку состояние знакового бита после выполнения оператора AND было известно.

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

Побитовые составные операторы с присваиванием

Подобно алгебраическим операторам, все двоичные побитовые операторы имеют составную форму, которая объединяет побитовый оператор с оператором присваивания. Например, следующие два оператора, выполняющие сдвиг вправо на четыре позиции в значении переменной a, эквивалентны.

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

Аналогично эквивалентны и следующие два оператора, которые присваивают переменной a результат выполнения побитовой операции a OR b.

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

Следующая программа создает несколько целочисленных переменных, а затем использует составные побитовые операторы с присваиванием для манипулирования этими переменными.

```
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);
    }
}
```

Эта программа создает следующий вывод.

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

Операторы сравнения

Операторы сравнения определяют отношение одного операнда с другим. В частности, они определяют равенство и порядок следования. Операторы сравнения перечислены в табл. 4.4.

Таблица 4.4. Операторы сравнения в Java

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

Результат выполнения этих операторов — логическое значение. Наиболее часто операции сравнения используют в выражениях, которые управляют оператором `if` и различными операторами цикла.

В Java можно сравнивать значения любых типов, в том числе целые значения, значения с плавающей точкой, символы и булевы значения, используя проверку равенства `==` и неравенства `!=`. Обратите внимание на то, что в Java равенство обозначают двумя знаками "равно", а не одним. (Одиночный знак "равно" — оператор присваивания.) Сравнение с помощью операторов упорядочения применимо только к числовым типам. То есть сравнение для определения того, какой из операндов больше или меньше другого, можно выполнять только для целочисленных операндов, операндов с плавающей точкой или символьных операндов.

Как уже отмечалось, результат оператора сравнения представляет собой логическое значение. Например, следующий фрагмент кода вполне допустим.

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

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

Те читатели, которые знакомы с языками C/C++, должны обратить внимание на следующее. В программах на языке C/C++ следующие типы операторов встречаются очень часто.

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

В программе Java эти операторы должны быть записаны следующим образом.

```
if(done == 0) ... // Это стиль Java.
if(done != 0) ...
```

Это обусловлено тем, что в языке Java определение значений "истинно" и "ложно" отличается от их определений в языках C/C++. В языке C/C++ истинным считается любое ненулевое значение, а ложным — нуль. В Java значения `true` (истинно) и `false` (ложно) — нечисловые значения, которые никак не сопоставимы с нулевым или ненулевым значением. Поэтому, чтобы сравнить значение с нулевым или ненулевым значением, необходимо явно использовать один или несколько операторов сравнения.

Логические операторы

Описанные в этом разделе логические операторы работают только с операндами типа `boolean`. Все логические операторы с двумя операндами объединяют два логических значения, образуя результирующее логическое значение. Логические операторы перечислены в табл. 4.5.

Таблица 4.5. Логические операторы в Java

Оператор	Описание
`&`	Логическое AND (И)
`|`	Логическое OR (ИЛИ)
`^`	Логическое XOR (исключающее OR (ИЛИ))
`||`	Сокращенное OR
`&&`	Сокращенное AND
`!`	Логическое унарное NOT (НЕ)
`&=`	AND с присваиванием
`|=`	OR с присваиванием
`^=`	XOR с присваиванием
`==`	Равно
`!=`	Не равно
`?:`	Троичный условный оператор

Логические операторы `&`, `|` и `^` действуют применительно к значениям типа `boolean` точно так же, как и по отношению к битам целочисленных значений. Логический оператор `!` инвертирует булево состояние: `!true == false` и `!false == true`. Результат выполнения каждого из логических операторов приведен в табл. 4.6.

Таблица 4.6. Результаты выполнения логических операторов

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 выполняет практически те же действия, что и пример программы `BitLogic`, представленный ранее, но она работает с логическими значениями типа `boolean`, а не с двоичными разрядами.

```
// Демонстрация применения булевых логических операторов.
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);
    }
}
```

Выполняя эту программу, легко убедиться, что к значениям типа `boolean` применяются те же логические правила, что и к битам. Как видно из следующего вывода, в Java строковое представление значения типа `boolean` — значение одного из литералов `true` или `false`.

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

Сокращенные логические операторы

Язык Java предоставляет два интересных логических оператора, которые не встречаются во многих других языках программирования. Это вторые версии булевых операторов AND и OR, обычно называемые сокращенны*ми* логическими операторами. Как видно из ранее приведенной таблицы, результат выполнения оператора OR равен `true`, когда значение операнда `A` равно `true`, независимо от значения операнда `B`. Аналогично результат выполнения оператора AND равен `false`, когда значение операнда `A` равно `false`, независимо от значения операнда `B`. При использовании форм `||` и `&&` этих операторов вместо `|` и `&` программа Java не будет вычислять значение правого операнда, если результат выражения можно определить по значению одного левого операнда. Это свойство очень удобно в тех случаях, когда значение правого операнда зависит от значения левого. Например, следующий фрагмент кода демонстрирует преимущество применения сокращенных логических операторов для выяснения допустимости операции деления перед вычислением ее результата.

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

Благодаря применению сокращенной формы оператора AND (&&) исключается риск возникновения исключения времени выполнения в случае равенства знаменателя (denom) нулю. Если бы эта строка кода была записана с применением одинарного символа & оператора AND, программа вычисляла бы обе части выражения, что приводило бы к исключению времени выполнения при равенстве значения переменной denom нулю.

Сокращенные формы операторов AND и OR принято применять в тех случаях, когда требуются операторы булевой логики, а их односимвольные версии используются исключительно для побитовых операций. Однако существуют исключения из этого правила. Например, рассмотрим следующий оператор.

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

В данном случае одиночный символ & гарантирует применение оператора инкремента к значению e независимо от равенства 1 значения c.

На заметку! Формальная спецификация языка Java называет сокращенные операторы *условным И* (conditional-end) и *условным ИЛИ* (conditional-or).

Оператор присваивания

Мы использовали оператор присваивания начиная с главы 2. Теперь пора рассмотреть этот оператор формально. Оператором *присваивания* служит одиночный знак равенства, =. В Java оператор присваивания работает аналогично тому, как и во многих компьютерных языках. Он имеет следующую общую форму.

```
переменная = выражение;
```

В этом операторе тип *переменная* должен соответствовать типу *выражение*.

Оператор присваивания имеет одну интересную особенность, с которой вы, возможно, еще не знакомы: он позволяет создавать цепочки присваиваний. Например, рассмотрим следующий фрагмент кода.

```
int x, y, z;
x = y = z = 100; // устанавливает значения
                 // переменных x, y и z равными 100
```

В этом фрагменте кода единственный оператор устанавливает значения трех переменных x, y и z равными 100. Это обусловлено тем, что оператор = использует значение правого выражения. Таким образом, значением выражения z=100 будет 100, оно и присваивается переменной y, а затем — переменной x. Использование "цепочки присваивания" — удобный способ установки общего значения группы переменных.

Оператор ?

Синтаксис Java содержит специальный *троичный условный оператор*, которым можно заменять определенные типы операторов if-then-else. Это — оператор ?. Вначале он может казаться несколько непонятным, но со временем вы убедитесь в его исключительной эффективности. Этот оператор имеет следующую общую форму.

```
выражение1 ? выражение2 : выражение3
```

Здесь *выражение1* — любое выражение, приводящее к значению типа `boolean`. Если значение *выражение1* — `true`, программа вычисляет значение *выражение2*. В противном случае программа вычисляет значение *выражение3*. Результат выполнения оператора ? равен значению вычисленного выражения. И *выражение2*, и *выражение3* должны возвращать значение одинакового (или совместимого) типа, которым не может быть тип `void`.

Ниже приведен пример применения этого оператора.

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

Когда программа Java вычисляет это выражение присваивания, вначале она проверяет выражение слева от знака вопроса. Если значение переменной `denom` равно 0, программа вычисляет выражение, указанное между знаками вопроса и двоеточия, и использует вычисленное значение в качестве значения всего выражения ?. Если значение переменной `denom` не равно 0, программа вычисляет выражение, указанное после двоеточия, и использует его в качестве значения всего выражения ?. Затем значение, полученное в результате выполнения оператора ?, присваивается переменной `ratio`.

Следующий пример программы демонстрирует применение оператора ?. Эта программа служит для получения абсолютного значения переменной.

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

        i = 10;
        k = i < 0 ? -i : i; // получение абсолютного значения i
        System.out.print("Абсолютное значение ");
        System.out.println(i + " равно " + k);

        i = -10;
        k = i < 0 ? -i : i; // получение абсолютного значения i
        System.out.print("Абсолютное значение ");
        System.out.println(i + " равно " + k);
    }
}
```

Эта программа создает следующий вывод.

```
Абсолютное значение 10 равно 10
Абсолютное значение -10 равно 10
```

Приоритет операторов

Приоритеты операторов Java, от высшего к низшему, описаны в табл. 4.7. Операторы, расположенные в том же ряду таблицы, имеют равный приоритет. Бинарные операторы имеют порядок вычисления слева направо (за исключением присваивания, которое обрабатывается справа налево). Хотя технически скобки [] и () являются разделителями, они способны действовать как операторы. В этом качестве они имеют самый высокий приоритет.

Таблица 4.7. Приоритеты операторов Java

Высший приоритет						
++ (постфиксный)	-- (постфиксный)					
++ (префиксный)	-- (префиксный)	~	!	+ (унарный)	- (унарный)	(*приведение типа*)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
=	op=					
Низший приоритет						

Использование круглых скобок

Круглые скобки повышают приоритет заключенных в них операторов. Часто это необходимо для получения требуемого результата. Например, рассмотрим следующее выражение.

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

Вначале это выражение добавляет 3 к значению переменной b, а затем сдвигает значение переменной a вправо на полученное количество позиций. Используя избыточные круглые скобки, это выражение можно было бы записать следующим образом.

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

Но если вначале нужно выполнить сдвиг значения переменной a вправо на b позиций, а затем добавить 3 к полученному результату, необходимо использовать круглые скобки так.

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

Кроме изменения обычного приоритета операторов, иногда круглые скобки можно использовать для облегчения понимания смысла выражения. Сложные выражения могут оказаться трудными для понимания. Добавление избыточных, но облегчающих понимание смысла выражения круглых скобок может способство-

вать исключению недоразумений в будущем. Например, какое из следующих выражений легче прочесть?

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

И еще один немаловажный момент: использование круглых скобок (избыточных или не избыточных) не ведет к снижению производительности программы. Поэтому добавление круглых скобок для повышения читабельности программы не оказывает на нее отрицательного влияния.

ГЛАВА 5 Управляющие операторы

В языках программирования управляющие операторы используются для реализации переходов и ветвлений в потоке выполнения команд программы в соответствии с ее состоянием. Управляющие операторы программы Java можно разделить на следующие категории: операторы выбора, операторы цикла и операторы перехода. Операторы *выбора* позволяют программе выбирать различные ветви выполнения команд в соответствии с результатом выражения или состоянием переменной. Операторы *цикла* позволяют программе повторять выполнение одного или нескольких операторов (т.е. они образуют циклы). Операторы *перехода* обеспечивают возможность нелинейного выполнения программы. В этой главе мы рассмотрим все управляющие операторы Java.

Операторы выбора

В языке Java поддерживаются два оператора выбора: `if` и `switch`. Эти операторы позволяют управлять порядком выполнения команд программы в соответствии с условиями, которые известны только во время выполнения. Читатели будут приятно удивлены возможностями и гибкостью этих двух операторов.

Оператор `if`

В этой главе мы подробно рассмотрим оператор `if`, изначально представленный в главе 2. Это – оператор условного выбора ветви программы Java. Его можно использовать для направления выполнения программы по двум различным ветвям. Общая форма этого оператора выглядит следующим образом.

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

Здесь каждый *оператор* – это одиночный оператор или составной оператор, заключенный в фигурные скобки (т.е. *блок*). *Условие* – это любое выражение, возвращающее значение типа `boolean`. Выражение `else` не обязательно.

Оператор `if` работает следующим образом: если *условие* истинно, программа выполняет *оператор1*. В противном случае она выполняет *оператор2* (если он существует). Ни в одной из ситуаций программа не будет выполнять оба оператора. Например, рассмотрим следующий фрагмент кода.

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

В данном случае, если значение переменной a меньше значения переменной b, значение переменной a устанавливается равным нулю. В противном случае значение переменной b устанавливается равным нулю. Ни в одной из ситуаций значения обеих переменных a и b не могут получить нулевые значения.

Чаще всего в управляющих выражениях оператора if будут использоваться операции сравнения. Но это не обязательно. Для управления оператором if можно применять и одиночную переменную типа boolean, как показано в следующем фрагменте кода.

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

Помните, что только один оператор может следовать непосредственно за ключевым словом if или else. Если нужно включить больше операторов, придется создать код, подобный приведенному в следующем фрагменте.

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

В этом примере оба оператора, помещенные внутри блока, будут выполняться, если значение переменной bytesAvailable больше нуля.

Некоторые программисты предпочитают использовать в операторе if фигурные скобки даже при наличии только одного оператора в каждом выражении. Это упрощает добавление операторов в дальнейшем и избавляет от беспокойства по поводу наличия фигурных скобок. Фактически пропуск определения блока в тех случаях, когда он необходим, – достаточно распространенная ошибка. Например, рассмотрим следующий фрагмент кода.

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

Если судить по уровню отступа в коде, предполагалось, что оператор bytesAvailable=n; должен выполняться внутри выражения else. Однако, как вы помните, в программе Java отступы значения не имеют, и компилятор никак не может "узнать" намерения программиста. Компиляция этого кода будет выполняться без вывода каких-либо предупреждающих сообщений, но при выполнении программа будет вести себя не так, как было задумано. В следующем фрагменте ошибка предыдущего примера исправлена.

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

```
} else
    waitForMoreData();
```

Вложенные операторы if

Вложенный оператор if — это оператор if, являющийся целью другого оператора if или else. В программах вложенные операторы if встречаются очень часто. При использовании вложенных операторов if важно помнить, что оператор else всегда связан с ближайшим оператором if, расположенным в одном с ним блоке и еще не связанным с другим оператором else.

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

Как видно из комментариев, последний оператор else связан не с оператором if(j<20), поскольку тот не находится в одном с ним блоке (несмотря на то, что он является ближайшим оператором if, который еще не связан с оператором else). Последний оператор else связан с оператором if(i==10). Внутренний оператор else связан с оператором if(k>100), поскольку тот является ближайшим внутри этого же блока.

Конструкция if-else-if

Распространенной конструкцией программирования, построенной на основе последовательности вложенных операторов if, является конструкция if-else-if. Она выглядит следующим образом.

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

Операторы if выполняются последовательно, сверху вниз. Как только одно из условий, управляющих оператором if, становится равным true, программа выполняет оператор, связанный с данным оператором if, и пропускает остальную часть конструкции if-else-if. Если ни одно из условий не выполняется (не равно true), программа выполнит заключительный оператор else. Этот последний оператор служит условием, используемым по умолчанию. Иными словами, если проверка всех остальных условий дает отрицательный результат, программа выполняет последний оператор else. Если заключительный оператор else не указан, а результат проверки всех остальных условий равен false, программа не будет выполнять никаких действий.

Ниже приведен пример программы, в которой конструкция if-else-if служит для определения времени года, к которому относится конкретный месяц.

```
// Демонстрация применения 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 = "зиме";
        else if(month == 3 || month == 4 || month == 5)
            season = "весне";
        else if(month == 6 || month == 7 || month == 8)
            season = "лету";
        else if(month == 9 || month == 10 || month == 11)
            season = "осени";
        else
            season = "вымышленным месяцам";
        System.out.println("Апрель относится к " + season + ".");
    }
}
```

Программа создает следующий вывод.

```
Апрель относится к весне
```

Прежде чем продолжить чтение главы, вы можете поэкспериментировать с этой программой. Убедитесь, что независимо от значения, присвоенного переменной `month`, внутри конструкции `if-else-if` будет выполняться только один оператор присваивания.

Оператор `switch`

Оператор `switch` – оператор ветвления в Java. Он предлагает простой способ направления потока выполнения команд по различным ветвям кода в зависимости от значения управляющего выражения. Зачастую он оказывается эффективнее применения длинных последовательностей операторов `if-else-if`. Общая форма оператора `switch` имеет следующий вид.

```
switch (выражение) {
    case значение1:
        // последовательность операторов
        break;
    case значение2:
        // последовательность операторов
        break;
...
...
...
    case значениеN:
        // последовательность операторов
        break;
    default:
        // последовательность операторов, выполняемая по умолчанию
}
```

Для всех версий Java до JDK 7 *выражение* должно иметь тип `type`, `short`, `int`, `char` или тип перечисления. (Перечисления описаны в главе 12.) Начиная с JDK 7 выражение может иметь также тип `String`. Каждое значение, определенное в операторах `case`, должно быть уникальным константным выражением (таким, как литеральное значение). Дублирование значений `case` не допускается. Тип каждого *значения* должен быть совместим с типом *выражения*.

Оператор `switch` работает следующим образом. Значение выражения сравнивается с каждым значением в операторах `case`. При обнаружении совпадения программа выполняет последовательность кода, следующую за данным оператором `case`. Если значения ни одной из констант не совпадают со значением вы-

ражения, программа выполняет оператор default. Однако этот оператор не обязателен. При отсутствии совпадений со значениями case и отсутствии оператора default программа не выполняет никаких дальнейших действий.

Оператор break внутри последовательности switch служит для прерывания последовательности операторов. Как только программа доходит до оператора break, она продолжает выполнение с первой строки кода, следующей за всем оператором switch. Этот оператор оказывает действие "выхода" из оператора switch.

Ниже представлен 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 единице.");
                    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 3.");
            }
    }
}
```

Эта программа создает следующий вывод.

```
i равно нулю.
i равно единице.
i равно двум.
i равно трем.
i больше 3.
i больше 3.
```

Как видите, на каждой итерации цикла программа выполняет операторы, связанные с константой case, которая соответствует значению переменной i. Все остальные операторы пропускаются. После того как значение переменной i становится больше 3, оно перестает соответствовать какому-либо значению case, поэтому программа выполняет оператор default.

Оператор break не обязателен. Если его опустить, программа продолжит выполнение со следующего оператора case. В некоторых случаях желательно использовать несколько операторов case без разделяющих их операторов break. Например, рассмотрим следующую программу.

```
// В последовательности 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 равно или больше 10");
            }
    }
}
```

Эта программа создает следующий 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 равно или больше 10
i равно или больше 10
```

Как видите, программа выполняет каждый оператор `case`, пока не достигнет оператора `break` (или конца оператора `switch`).

Хотя приведенный пример был создан специально в качестве иллюстрации, пропуск операторов `break` находит множество применений в реальных программах. В качестве более реалистичного примера рассмотрим измененную версию приведенной ранее программы со временами года. В этой версии мы использовали оператор `switch`, что позволило создать более эффективную реализацию.

```
// Усовершенствованная версия программы с временами года.
class Switch {
    public static void main(String args[]) {
        int month = 4;
        String season;
        switch (month) {
            case 12:
            case 1:
            case 2:
                season = "зиме";
                break;
            case 3:
            case 4:
            case 5:
                season = "весне";
                break;
            case 6:
            case 7:
            case 8:
                season = "лету";
                break;
            case 9:
```

```
            case 10:
            case 11:
                season = "осени";
                break;
            default:
                season = "вымышленным месяцам";
        }
        System.out.println("Апрель относится к" + season + ".");
    }
}
```

Как упоминалось, начиная с JDK 7 вы можете использовать строку для контроля оператора `switch`.

```
// Использование строки для контроля оператора switch.
class StringSwitch {
    public static void main(String args[]) {
        String str = "два";
        switch(str) {
            case "один":
                System.out.println("один");
                break;
            case "два":
                System.out.println("два");
                break;
            case "три":
                System.out.println("три");
                break;
        default:
                System.out.println("нет соответствия)");
                break;
        }
    }
}
```

Как вы и ожидали, вывод будет следующим.

```
два
```

Строка, содержащаяся в переменной `str` (которой в данном случае является `"два"`), сравнивается с константами оператора `case`. Когда находится соответствие (во втором операторе `case`), выполняется связанная с ним последовательность кода.

Возможность использовать строки в операторе `switch` упрощает много ситуаций. Например, использование оператора `switch` со строками является существенным усовершенствованием по сравнению с использованием эквивалентной последовательности операторов `if/else`. Однако выбор среди строк дороже выбора среди целых чисел. Поэтому лучше применять строки только в тех случаях, когда контролирующие данные уже находятся в строковой форме. Другими словами, не используйте строки в операторе `switch` без необходимости.

Вложенные операторы `switch`

Оператор `switch` можно использовать в последовательности операторов внешнего оператора `switch`. Такой оператор называют *вложенным* оператором `switch`. Поскольку оператор `switch` определяет собственный блок, каких-либо конфликтов между константами `case` внутреннего и внешнего операторов `switch` не происходит. Например, следующий фрагмент полностью допустим.

```
switch(count) {
    case 1:
        switch(target) {// вложенный оператор switch
            case 0:
                System.out.println("target равна 0");
                break;
            case 1: // конфликты с внешним оператором switch отсутствуют
                System.out.println("target равна 1");
                break;
        }
        break;
    case 2: // ...
```

В данном случае оператор `case1:` внутреннего оператора `switch` не конфликтует с оператором `case1:` внешнего оператора `switch`. Программа сравнивает значение переменной `count` только со списком ветвей `case` внешнего уровня. Если значение `count` равно 1, программа сравнивает значение переменной `target` с внутренним списком ветвей `case`.

В результате можно выделить следующие три важных свойства оператора `switch`.

- Оператор `switch` отличается от оператора `if` тем, что он может выполнять проверку только равенства, в то время как оператор `if` может вычислять результат булевого выражения любого типа. То есть оператор `switch` ищет только соответствие между значением выражения и одной из констант `case`.
- Никакие две константы `case` в одном и том же операторе `switch` не могут иметь одинаковые значения. Конечно, внутренний оператор `switch` и содержащий его внешний оператор `switch` могут иметь одинаковые константы `case`.
- Как правило, оператор `switch` эффективнее набора вложенных операторов `if`.

Последнее свойство представляет особый интерес, поскольку позволяет понять работу компилятора Java. Компилируя оператор `switch`, компилятор Java будет проверять каждую константу `case` и создавать "таблицу переходов", которую будет использовать для выбора ветви программы в зависимости от значения выражения. Поэтому в тех случаях, когда требуется осуществлять выбор в большой группе значений, оператор `switch` будет выполняться значительно быстрее последовательности операторов `if-else`. Это обусловлено тем, что компилятору известно, что все константы `case` имеют один и тот же тип, и их нужно просто проверять на равенство значению выражения `switch`. Компилятор не располагает подобными сведениями о длинном списке выражений оператора `if`.

Операторы цикла

Операторами цикла Java являются `for`, `while` и `do-while`. Эти операторы образуют конструкции, обычно называемые *циклами*. Как, наверняка, известно читателям, циклы многократно выполняют один и тот же набор инструкций до тех пор, пока не будет удовлетворено условие завершения цикла. Как вы вскоре убедитесь, Java предлагает средства создания циклов, способные удовлетворить любые потребности программирования.

Цикл while

Оператор цикла while – наиболее часто используемый оператор цикла Java. Он повторяет оператор или блок операторов до тех пор, пока значение его управляющего выражения истинно. Он имеет следующую общую форму.

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

Здесь *условие* – любое булево выражение. Тело цикла будет выполняться до тех пор, пока условное выражение истинно. Когда *условие* становится ложным, управление передается строке кода, непосредственно следующей за циклом. Фигурные скобки могут быть опущены, только если в цикле повторяется лишь один оператор.

В качестве примера рассмотрим цикл while, который выполняет обратный отсчет начиная с 10; вывод – ровно 10 строк "тактов".

```
// Демонстрация использования цикла while.
class While {
    public static void main(String args[]) {
        int n = 10;
        while(n > 0) {
            System.out.println("такт " + n);
            n--;
        }
    }
}
```

После запуска эта программа выводит десять "тактов".

```
такт 10
такт 9
такт 8
такт 7
такт 6
такт 5
такт 4
такт 3
такт 2
такт 1
```

Поскольку цикл while вычисляет свое условное выражение в начале цикла, тело цикла не будет выполнено ни разу, если в самом начале условие оказывается ложным. Например, в следующем фрагменте кода метод println() никогда не будет вызван.

```
int a = 10, b = 20;
while(a > b)
    System.out.println("Эта строка отображаться не будет");
```

Тело цикла while (или любого другого цикла Java) может быть пустым. Это обусловлено тем, что синтаксис Java допускает применение *пустого оператора* (содержащего только символ "точка с запятой"). Например, рассмотрим следующую программу.

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

        i = 100;
```

```
        j = 200;

        // вычисление среднего значения i и j
        while(++i < --j) ; // в этом цикле тело цикла отсутствует
        System.out.println("Среднее значение равно " + i);
    }
}
```

Эта программа вычисляет среднее значение переменных `i` и `j`. Она создает следующий вывод.

```
Среднее значение равно 150
```

Этот цикл `while` работает следующим образом. Значение переменной `i` увеличивается, а значение переменной `j` уменьшается на единицу. Затем программа сравнивает эти два значения. Если новое значение переменной `i` по-прежнему меньше нового значения переменной `j`, цикл повторяется. Если значение переменной `i` равно значению переменной `j` или больше его, выполнение цикла прекращается. По выходу из цикла переменная `i` будет содержать среднее значение исходных значений переменных `i` и `j`. (Конечно, эта процедура работает только в том случае, если в самом начале значение переменной `i` меньше значения переменной `j`.) Как видите, никакой потребности в наличии тела цикла не существует. Все действия выполняются внутри самого условного выражения. В профессионально написанном коде Java короткие циклы часто не содержат тела, если само по себе управляющее выражение может выполнять все необходимые действия.

Цикл `do-while`

Как вы видели, если в начальный момент условное выражение, управляющее циклом `while`, ложно, тело цикла вообще не будет выполняться. Однако иногда желательно выполнить тело цикла хотя бы один раз, даже если в начальный момент условное выражение ложно. Иначе говоря, существуют ситуации, когда проверку условия прерывания цикла желательно выполнять в конце цикла, а не в его начале. К счастью, Java поддерживает именно такой цикл: `do-while`. Этот цикл всегда выполняет тело цикла хотя бы один раз, поскольку его условное выражение проверяется в конце цикла. Общая форма цикла `do-while` следующая.

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

При каждом повторении цикла `do-while` программа вначале выполняет тело цикла, а затем вычисляет условное выражение. Если это выражение истинно, цикл повторяется. В противном случае выполнение цикла прерывается. Как и во всех циклах Java, *условие* должно быть булевым.

Ниже приведена измененная программа вывода тактов, которая демонстрирует использование цикла `do-while`. Она создает такой же вывод, что и предыдущая версия.

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

        do {
            System.out.println("такт " + n);
            n--;
```

```
        } while(n > 0);
    }
}
```

Хотя с технической точки зрения в приведенной программе цикл записан правильно, его можно переписать в более эффективном виде.

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

В этом примере декремент переменной n и сравнение результирующего значения с нулем объединены в одном выражении `(--n>0)`. Программа работает следующим образом. Вначале она выполняет оператор `--n`, уменьшая значение переменной n на единицу и возвращая новое значение переменной n. Затем программа сравнивает это значение с нулем. Если оно больше нуля, выполнение цикла продолжается. В противном случае цикл прерывается.

Цикл `do-while` особенно удобен при выборе пункта меню, поскольку обычно в этом случае требуется, чтобы тело цикла меню выполнялось, по меньшей мере, один раз. Рассмотрим следующую программу, которая реализует очень простую систему справки по операторам выбора и цикла Java.

```
// Использование цикла do-while для выбора пункта меню
class Menu {
    public static void main(String args[])
        throws java.io.IOException {
        char choice;
        do {
            System.out.println("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 интересующий пункт:");
            choice = (char) System.in.read();
        } while( choice < '1' || choice > '5');

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

        switch(choice) {
            case '1':
                System.out.println("if:\n");
                System.out.println("if(условие) оператор;");
                System.out.println("else оператор;");
                break;
            case '2':
                System.out.println("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("while:\n");
                System.out.println("while(условие) оператор;");
                break;
```

```
            case '4':
                System.out.println("do-while:\n");
                System.out.println("do {");
                System.out.println("    оператор;");
                System.out.println("} while (условие);");
                break;
            case '5':
                System.out.println("for:\n");
                System.out.print("for(инициализация; условие; повторение)");
                System.out.println(" оператор;");
                break;
        }
    }
}
```

Пример вывода этой программы выглядит следующим образом.

```
Справка по:
    1. if
    2. switch
    3. while
    4. do-while
    5. for
Выберите интересующий пункт:
4
do-while:
do {
    оператор;
} while (условие);
```

В этой программе в цикле `do-while` осуществляется проверка допустимости введенного пользователем значения. Если это значение недопустимо, программа предлагает пользователю повторить ввод. Поскольку меню должно отобразиться, по меньшей мере, один раз, цикл `do-while` является прекрасным средством решения этой задачи.

Отметим еще несколько особенностей приведенного примера. Обратите внимание на то, что для считывания символов с клавиатуры используется метод `System.in.read()`. Это — одна из функций консольного ввода Java. Подробно методы консольного ввода-вывода рассмотрим в главе 13, а пока отметим, что в данном случае метод `System.in.read()` используется для выяснения сделанного пользователем выбора. Этот метод считывает символы из стандартного устройства ввода (возвращаемые в виде целочисленных значений — именно потому тип возвращаемого значения был приведен к типу `char`). По умолчанию данные из стандартного ввода помещаются в буфер построчно, поэтому, чтобы любые введенные символы были пересланы программе, необходимо нажать клавишу <Enter>.

Консольный ввод Java может вызывать некоторые затруднения при работе. Более того, большинство реальных программ Java будут графическими и ориентированными на работу в оконном режиме. Поэтому в данной книге консольному вводу уделяется не очень много внимания. Однако в данном случае он удобен. Еще один важный момент. Поскольку мы используем метод `System.in.read()`, программа должна содержать выражение `throws java.io.IOException`. Эта строка необходима для обработки ошибок ввода. Она является составной частью системы обработки исключений Java, которая будет рассмотрена в главе 10.

Цикл for

Простая форма цикла for была представлена в главе 2. Вскоре читатели убедятся в больших возможностях и гибкости этой конструкции.

Начиная с версии JDK 5 в Java существует две формы цикла for. Первая – традиционная форма, используемая начиная с исходной версии Java. Вторая – новая форма "for-each". Мы рассмотрим оба типа цикла for, начиная с традиционной формы.

Общая форма традиционного оператора цикла for выглядит следующим образом.

```
for(инициализация; условие; повторение) {
    // тело
}
```

Если в цикле будет повторяться только один оператор, фигурные скобки можно опустить.

Цикл for действует следующим образом. При первом запуске цикла программа выполняет *инициализационную* часть цикла. В общем случае это выражение, устанавливающее значение *управляющей переменной цикла*, которая действует в качестве счетчика, управляющего циклом. Важно понимать, что выражение инициализации выполняется только один раз. Затем программа вычисляет *условие*, которое должно быть булевым выражением. Как правило, выражение сравнивает значение управляющей переменной с целевым значением. Если это значение истинно, программа выполняет тело цикла. Если оно ложно, выполнение цикла прерывается. Затем программа выполняет часть *повторение* цикла. Обычно это выражение, которое увеличивает или уменьшает значение управляющей переменной. Затем программа повторяет цикл, при каждом прохождении вначале вычисляя условное выражение, затем выполняя тело цикла и выражение повторения. Процесс повторяется до тех пор, пока значение выражения повторения не станет ложным.

Ниже приведена версия программы подсчета "тактов", в которой использован цикл for.

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

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

Объявление управляющих переменных цикла внутри цикла for

Часто переменная, которая управляет циклом for, требуется только для него и не используется нигде больше. В этом случае переменную можно объявить внутри инициализационной части оператора for. Например, предыдущую программу можно переписать, объявляя управляющую переменную n типа int внутри цикла for.

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

        // в данном случае переменная n объявляется внутри цикла for
        for(int n=10; n>0; n--)
            System.out.println("такт " + n);
    }
}
```

При объявлении переменной внутри цикла `for` необходимо помнить о следующем: область и продолжительность существования этой переменной полностью совпадают с областью и продолжительностью существования оператора `for`. (То есть область существования переменной ограничена циклом `for`.) Вне цикла `for` переменная прекратит свое существование. Если управляющую переменную цикла нужно использовать в других частях программы, ее нельзя объявлять внутри цикла `for`.

В тех случаях, когда управляющая переменная цикла не требуется нигде больше, большинство программистов Java предпочитают объявлять ее внутри оператора `for`. В качестве примера приведем простую программу, которая проверяет, является ли введенное число простым. Обратите внимание на то, что управляющая переменная цикла `i` объявлена внутри цикла `for`, поскольку она нигде больше не требуется.

```
// Проверка принадлежности к простым числам.
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("Простое");
        else System.out.println("Не простое");
    }
}
```

Использование запятой

В ряде случаев требуется указание нескольких операторов в инициализационной и итерационной частях цикла `for`. Например, рассмотрим 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, управление этим циклом осуществляется одновременно двумя переменными. Поскольку цикл управляется двумя переменными, желательно, чтобы их обе можно было бы включить в сам оператор `for`, а не выполнять обработку

переменной b вручную. К счастью, язык Java предоставляет средства для выполнения этой задачи. Чтобы две или более переменных могли управлять циклом for, Java позволяет указывать по нескольку операторов как в инициализационной, так и итерационной частях оператора for. Один от другого операторы отделяются запятыми.

Используя запятые, предыдущий цикл for можно записать в более эффективном виде.

```
// 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);
        }
    }
}
```

В этом примере в инициализационной части цикла мы устанавливаем начальные значения обеих управляющих переменных a и b. Оба разделенных запятой оператора в итерационной части выполняются при каждом повторении цикла. Программа создает следующий вывод.

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

На заметку! Читатели, знакомые с языками C/C++, знают, что в этих языках запятая — оператор, который можно использовать в любом допустимом выражении. Однако в Java это не так. Здесь запятая служит разделителем.

Разновидности цикла for

Цикл for имеет несколько разновидностей, которые увеличивают его возможности и повышают применимость. Гибкость этого цикла обусловлена тем, что его три части — инициализационную, проверку условий и итерационную — не обязательно использовать только по прямому назначению. Фактически каждый раздел оператора for можно применять в любых целях. Рассмотрим несколько примеров.

Одна из наиболее часто встречающихся вариаций предполагает использование условного выражения. В частности, это выражение не обязательно должно выполнять сравнение управляющей переменной цикла с каким-либо целевым значением. Фактически условием, управляющим циклом for, может быть любое булево выражение. Например, рассмотрим следующий фрагмент.

```
boolean done = false;

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

В этом примере выполнение цикла for продолжается до тех пор, пока значение переменной done не будет установлено равным true. В этом цикле проверка значения управляющей переменной цикла i не выполняется.

Ниже приведена еще одна интересная разновидность цикла `for`. Инициализационное или итерационное выражения либо оба могут отсутствовать, как показано 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++;
        }
    }
}
```

В этом примере инициализационное и итерационное выражения вынесены за пределы цикла `for`. В результате соответствующие части оператора `for` пусты. Хотя в этом простом примере — действительно, его можно считать достаточно примитивным — это и не имеет никакого значения, в отдельных случаях такой подход имеет смысл. Например, если начальное условие определяется сложным выражением где-то в другом месте программы или значение управляющей переменной цикла изменяется случайным образом в зависимости от действий, выполняемых внутри тела цикла, вполне целесообразно оставить эти части цикла `for` пустыми.

Приведем еще одну разновидность цикла `for`. Оставляя все три части оператора пустыми, можно умышленно создать бесконечный цикл (цикл, который никогда не завершается).

```
for( ; ; ) {
    // ...
}
```

Этот цикл может выполняться бесконечно, поскольку условие, по которому он был бы прерван, отсутствует. Хотя некоторые программы, такие как командные процессоры операционной системы, требуют наличия бесконечного цикла, большинство "бесконечных циклов" в действительности представляет собой всего лишь циклы с особыми условиями прерывания. Как вы вскоре убедитесь, существует способ прерывания цикла (даже бесконечного, подобного приведенному примеру), который не требует использования обычного условного выражения цикла.

Версия "for-each" цикла `for`

Начиная с версии JDK 5 в Java можно использовать вторую форму цикла `for`, реализующую цикл в стиле "for-each" ("для каждого"). Как вам, возможно, известно, в современной теории языков программирования все большее применение находит концепция циклов "for-each", которые быстро становятся стандартными функциональными возможностями во многих языках. Цикл в стиле "for-each" предназначен для строго последовательного выполнения повторяющихся действий по отношению к коллекции объектов, такой как массив. В отличие от некоторых языков, подобных C#, в котором для реализации циклов "for-each" используют ключевое слово `foreach`, в Java возможность применения цикла "for-each" реализована за счет усовершенствования цикла `for`. Преимущество этого подхода состоит в том, что для его реализации не требуется дополнительное ключевое слово и никакой ранее существовавший код не разрушается. Цикл `for` в стиле "for-each"

называют также *усовершенствованным* циклом `for`. Общая форма версии "for-each" цикла `for` имеет следующий вид.

```
for(тип итер_пер : коллекция) блок_операторов
```

Здесь *тип* указывает тип, а *итер_пер* — имя *итерационной переменной*, которая последовательно будет принимать значения из коллекции, от первого до последнего. Элемент *коллекция* указывает коллекцию, по которой должен выполняться цикл. С циклом `for` можно применять различные типы коллекций, но в этой главе мы будем использовать только массивы. (Другие типы коллекций, которые можно применять с циклом `for`, вроде определенных в инфраструктуре коллекций Collection Framework, рассматриваются в последующих главах книги.) На каждой итерации цикла программа извлекает следующий элемент коллекции и сохраняет его в переменной *итер_пер*. Цикл выполняется до тех пор, пока не будут получены все элементы коллекции.

Поскольку итерационная переменная получает значения из коллекции, *тип* должен совпадать (или быть совместимым) с типом элементов, хранящихся в коллекции. Таким образом, при переборе массива *тип* должен быть совместим с типом элемента массива.

Чтобы понять побудительные причины применения циклов в стиле "for-each", рассмотрим тип цикла `for`, для замены которого предназначен этот стиль. В следующем фрагменте для вычисления суммы значений элементов массива 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];
```

Чтобы вычислить сумму, мы последовательно считываем значения каждого элемента массива `nums`. Таким образом, чтение всего массива выполняется строго последовательно. Это осуществляется за счет индексации массива `nums` вручную по управляющей переменной цикла `i`.

Цикл `for` в стиле "for-each" позволяет автоматизировать этот процесс. В частности, применение такого цикла позволяет не устанавливать значение счетчика цикла за счет указания его начального и конечного значений и исключает необходимость индексации массива вручную. Вместо этого программа автоматически выполняет цикл по всему массиву, последовательно получая значения каждого из его элементов, от первого до последнего. Например, с учетом версии "for-each" цикла `for` предыдущий фрагмент можно переписать следующим образом.

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int x: nums) sum += x;
```

При каждом прохождении цикла, переменной `x` автоматически присваивается значение, равное значению следующего элемента массива `nums`. Таким образом, на первой итерации переменная `x` содержит 1, на второй — 2 и т.д. При этом не только упрощается синтаксис программы, но и исключается возможность ошибок выхода за пределы массива.

Ниже показан пример полной программы, иллюстрирующей применение описанной версии "for-each" цикла `for`.

```
// 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;

        // использование стиля for-each для отображения и
        // суммирования значений
        for(int x : nums) {
            System.out.println("Значение равно: " + x);
            sum += x;
        }

        System.out.println("Сумма равна: " + sum);
    }
}
```

Эта программа создает следующий вывод.

```
Значение равно: 1
Значение равно: 2
Значение равно: 3
Значение равно: 4
Значение равно: 5
Значение равно: 6
Значение равно: 7
Значение равно: 8
Значение равно: 9
Значение равно: 10
Сумма равна: 55
```

Как видно из этого вывода, оператор `for` в стиле "for-each" автоматически перебирает элементы массива, от наименьшего индекса к наибольшему.

Хотя повторение цикла `for` в стиле "for-each" выполняется до тех пор, пока не будут обработаны все элементы массива, цикл можно прервать и раньше, используя оператор `break`. Например, следующая программа суммирует значения пяти первых элементов массива `nums`.

```
// Использование оператора break в цикле 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 };

        // использование цикла for для отображения и
        // суммирования значений
        for(int x : nums) {
            System.out.println("Значение равно: " + x);
            sum += x;
            if(x == 5) break; // прекращение цикла после
                              // получения 5 значений
        }
        System.out.println("Сумма пяти первых элементов равна: " + sum);
    }
}
```

Программа создает следующий вывод.

```
Значение равно: 1
Значение равно: 2
Значение равно: 3
Значение равно: 4
Значение равно: 5
Сумма пяти первых элементов равна: 15
```

Как видите, выполнение цикла прекращается после получения значения пятого элемента. Оператор `break` можно использовать также и с другими циклами Java. Подробнее этот оператор будет рассмотрен в последующих разделах.

При использовании цикла в стиле "for-each" необходимо помнить о следующем: его итерационная переменная является переменной "только для чтения", поскольку она связана только с исходным массивом. Оператор присваивания значения итерационной переменной не оказывает никакого влияния на исходный массив. Иначе говоря, содержимое массива нельзя изменять, присваивая новое значение итерационной переменной. Например, рассмотрим следующую программу.

```
// Переменная цикла 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; // этот оператор не оказывает никакого
                        // влияния на массив nums
        }

        System.out.println();

        for(int x : nums)
            System.out.print(x + " ");
        System.out.println();
    }
}
```

Первый цикл `for` увеличивает значение итерационной переменной на 10. Однако этот оператор присваивания не оказывает никакого влияния на исходный массив `nums`, как видно из результата выполнения второго оператора `for`. Создаваемый программой вывод подтверждает сказанное.

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

Итерация в многомерных массивах

Усовершенствованная версия цикла `for` применима также и к многомерным массивам. Однако следует помнить, что в языке Java многомерные массивы состоят из *массивов массивов*. (Например, двухмерный массив — это массив одномерных массивов.) Это важно при переборе многомерного массива, поскольку результат каждой итерации — *следующий массив*, а не отдельный элемент. Более того, тип итерационной переменной цикла `for` должен быть совместим с типом получаемого массива. Например, в случае двухмерного массива итерационная переменная должна быть ссылкой на одномерный массив. В общем случае при использовании цикла "for-each" для перебора массива размерностью N получаемые объекты будут массивами размерностью $N-1$. Чтобы понять, что из этого следует, рассмотрим следующую программу. В ней вложенные циклы `for` служат для получения упорядоченных по строкам элементов двухмерного массива.

```
// Использование цикла for в стиле for-each применительно
// к двухмерному массиву.
class ForEach3 {
    public static void main(String args[]) {
        int sum = 0;
```

```
        int nums[][] = new int[3][5];

        // присвоение значений элементам массива nums
        for(int i = 0; i < 3; i++)
            for(int j = 0; j < 5; j++)
                nums[i][j] = (i+1)*(j+1);

        // использование цикла for в стиле for-each для отображения
        // и суммирования значений
        for(int x[] : nums) {
            for(int y : x) {
                System.out.println("Значение равно: " + y);
                sum += y;
            }
        }
        System.out.println("Сумма: " + sum);
    }
}
```

Эта программа создает следующий вывод.

```
Значение равно: 1
Значение равно: 2
Значение равно: 3
Значение равно: 4
Значение равно: 5
Значение равно: 2
Значение равно: 4
Значение равно: 6
Значение равно: 8
Значение равно: 10
Значение равно: 3
Значение равно: 6
Значение равно: 9
Значение равно: 12
Значение равно: 15
Сумма: 90
```

Следующая строка этой программы заслуживает особого внимания.

```
for(int x[] : nums) {
```

Обратите внимание на способ объявления переменной `x`. Эта переменная — ссылка на одномерный массив целочисленных значений. Это необходимо, потому что результат выполнения каждой итерации цикла `for` — следующий *массив* в массиве `nums`, начиная с массива, указанного элементом `nums[0]`. Затем внутренний цикл `for` перебирает каждый из этих массивов, отображая значения каждого элемента.

Использование усовершенствованного цикла `for`

Поскольку каждый оператор `for` в стиле "for-each" может перебирать элементы массива только последовательно, начиная с первого и заканчивая последним, может показаться, что его применение ограничено. Но это не так. Множество алгоритмов требуют использования именно этого механизма. Одним из наиболее часто используемых алгоритмов является поиск. Например, следующая программа использует цикл `for` для поиска значения в неупорядоченном массиве. Поиск прекращается после обнаружения искомого значения.

```
// Поиск в массиве с применением 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;

        // использование цикла for в стиле for-each для
        // поиска в nums значения val
        for(int x : nums) {
            if(x == val) {
                found = true;
                break;
            }
        }

        if(found)
            System.out.println("Значение найдено!");
    }
}
```

В данном случае выбор стиля "for-each" для цикла `for` полностью оправдан, поскольку поиск в неупорядоченном массиве предполагает последовательный просмотр каждого элемента. (Конечно, если бы массив был упорядоченным, можно было бы использовать бинарный поиск, реализация которого требовала бы применения цикла другого стиля.) К другим типам приложений, которым применение циклов в стиле "for-each" предоставляет преимущества, относятся вычисление среднего значения, отыскание минимального или максимального значения в наборе, поиск дубликатов и т.п.

Хотя в примерах этой главы мы использовали массивы, цикл `for` в стиле "for-each" особенно удобен при работе с коллекциями, определенными в инфраструктуре Collections Framework (Инфраструктура коллекций), которая описана в части II. В более общем случае оператор `for` может перебирать элементы любой коллекции объектов, если эта коллекция удовлетворяет определенному набору ограничений, который описан в главе 17.

Вложенные циклы

Подобно другим языкам программирования, Java допускает использование вложенных циклов. Другими словами, один цикл может выполняться внутри другого. Например, в следующей программе использованы 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();
        }
    }
}
```

Эта программа создает следующий вывод.

```
..........
.........
........
.......
......
.....
```

```
....
...
..
.
```

Операторы перехода

В языке Java определены три оператора перехода: `break`, `continue` и `return`. Они передают управление другой части программы. Рассмотрим каждый из них.

На заметку! Кроме операторов перехода, рассмотренных в этом разделе, язык Java поддерживает еще один способ изменения порядка выполнения инструкций программы: обработку исключений. Обработка исключений предоставляет структурированный метод, используя который программа может обнаруживать и обрабатывать ошибки времени выполнения. Для поддержки этого метода служат ключевые слова `try`, `catch`, `throws` и `finally`. По сути, механизм обработки ошибок позволяет программе выполнять нелокальные ветви. Поскольку тема обработки исключений очень обширна, она рассмотрена в посвященной ей главе 10.

Использование оператора `break`

В языке Java оператор `break` находит три применения. Во-первых, как уже было показано, он завершает последовательность операторов в операторе `switch`. Во-вторых, его можно использовать для выхода из цикла. И в-третьих, этот оператор можно применять в качестве "цивилизованной" формы оператора безусловного перехода ("goto"). Рассмотрим последние два применения.

Использование оператора `break` для выхода из цикла

Используя оператор `break`, можно вызвать немедленное завершение цикла, пропуская условное выражение и любой остальной код в теле цикла. Когда программа встречает оператор `break` внутри цикла, она прекращает выполнение цикла и управление передается оператору, следующему за циклом. Ниже показан простой пример.

```
// Использование оператора break для выхода из цикла.
class BreakLoop {
    public static void main(String args[]) {
        for(int i=0; i<100; i++) {
            if(i == 10) break; // выход из цикла если i равно 10
            System.out.println("i: " + i);
        }
        System.out.println("Цикл завершен.");
    }
}
```

Эта программа создает следующий вывод.

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Цикл завершен.
```

Как видите, хотя цикл for должен был бы выполняться для значений управляющей переменной от 0 до 99, оператор break приводит к более раннему выходу из него, когда значение переменной i становится равным 10.

Оператор break можно использовать в любых циклах Java, в том числе в преднамеренно бесконечных циклах. Например, в предыдущей программе можно было применить цикл while. Эта программа создает вывод, совпадающий с предыдущим.

```
 / Использование оператора break для выхода из цикла while.
class BreakLoop2 {
    public static void main(String args[]) {
        int i = 0;

        while(i < 100) {
            if(i == 10) break; // выход из цикла, если i равно 10
            System.out.println("i: " + i);
            i++;
        }
        System.out.println("Цикл завершен.");
    }
}
```

В случае его использования внутри набора вложенных циклов оператор break осуществляет выход только из самого внутреннего цикла.

```
// Использование оператора break во вложенных циклах.
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; // выход из цикла, если j равно 10
                System.out.print(j + " ");
            }
            System.out.println();
        }
        System.out.println("Циклы завершены.");
    }
}
```

Эта программа создает 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
Циклы завершены.
```

Как видите, оператор break во внутреннем цикле может приводить к выходу только из этого цикла. На внешний цикл он не оказывает никакого влияния.

При использовании оператора break необходимо помнить следующее. Во-первых, в цикле можно использовать более одного оператора break. Однако при этом следует соблюдать осторожность. Как правило, применение слишком большого количества операторов break приводит к нарушению структуры кода. Во-вторых, оператор break, который завершает последовательность операторов в операторе switch, оказывает влияние только на данный оператор switch, а не на какие-либо содержащие его циклы.

Помните! Оператор break не был задуман в качестве обычного средства выхода из цикла. Для этого служит условное выражение цикла. Этот оператор следует использовать для выхода из цикла только в особых ситуациях.

Использование оператора break в качестве формы оператора безусловного перехода

Кроме применения с операторами switch и циклами, оператор break можно использовать и сам по себе в качестве "цивилизованной" формы оператора безусловного перехода ("goto"). Язык Java не содержит оператора "goto", поскольку он позволяет выполнять ветвление программ произвольным и неструктурированным образом. Как правило, код, который управляется оператором "goto", труден для понимания и поддержки. Кроме того, этот оператор исключает возможность оптимизации кода для определенного компилятора. Однако в некоторых случаях оператор "goto" — ценная и вполне допустимая конструкция управления потоком команд. Например, оператор "goto" может быть полезен при выходе из набора вложенных циклов с большим количеством уровней. Для таких ситуаций Java определяет расширенную форму оператора break. Используя эту форму, можно, например, осуществлять выход из одного или нескольких блоков кода. Эти блоки не обязательно должны быть частью цикла или оператора switch. Они могут быть любым блоком. Более того, можно точно указать оператор, с которого будет продолжено выполнение программы, поскольку эта форма оператора break работает с метками. Как будет показано, оператор break предоставляет все преимущества оператора "goto", не порождая его проблемы. Общая форма оператора break с меткой имеет следующий вид.

```
break метка;
```

Чаще всего *метка* — это имя метки, идентифицирующей блок кода. Им может быть как самостоятельный блок кода, так и целевой блок другого оператора. При выполнении этой формы оператора break управление передается названному блоку кода. Помеченный блок кода должен содержать оператор break, но он не обязательно должен быть непосредственно содержащим его блоком. В частности это означает, что оператор break с меткой можно применять для выхода из набора вложенных блоков. Однако его нельзя использовать для передачи управления внешнему блоку кода, который не содержит данный оператор break.

Чтобы пометить блок, необходимо поместить метку в его начале. *Метка* — это любой допустимый идентификатор Java, за которым следует двоеточие. Как только блок помечен, его метку можно использовать в качестве цели оператора break. В результате выполнение программы будет продолжено с *конца* помеченного блока. Например, следующая программа содержит три вложенных блока, каждый из которых помечен своей меткой. Оператор break приводит к переходу к концу блока с меткой second с пропуском двух вызовов метода println().

```
// Использование оператора break в качестве цивилизованной формы оператора
goto.
class Break {
  public static void main(String args[]) {
    boolean t = true;

    first: {
      second: {
        third: {
          System.out.println("Предшествует оператору break.");
          if(t) break second; // выход из блока second
            System.out.println("Этот оператор не будет выполняться");
        }
        System.out.println("Этот оператор не будет выполняться");
      }
```

```
      System.out.println("Этот оператор следует за блоком second.");
    }
  }
}
```

Эта программа создает следующий вывод.

```
Предшествует оператору break.
Этот оператор следует за блоком second.
```

Одно из наиболее распространенных применений оператора `break` с меткой — его использование для выхода из вложенных циклов. Например, в следующей программе внешний цикл выполняется только один раз.

```
// Использование оператора 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; // выход из обоих циклов
                System.out.print(j + " ");
            }
            System.out.println("Эта строка не будет выводиться");
        }
        System.out.println("Циклы завершены.");
    }
}
```

Эта программа создает следующий вывод.

```
Проход 0: 0 1 2 3 4 5 6 7 8 9 Циклы завершены.
```

Как видите, когда внутренний цикл выполняет выход во внешний цикл, это приводит к завершению обоих циклов.

Следует иметь в виду, что нельзя выполнить переход к какой-либо метке, которая определена не для содержащего данный оператор `break` блока. Например, следующая программа содержит ошибку, и поэтому ее компиляция будет невозможна.

```
// Эта программа содержит ошибку.
class BreakErr {
    public static void main(String args[]) {

        one: for(int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
        }

        for(int j=0; j<100; j++) {
            if(j == 10) break one; // ОШИБКА!
            System.out.print(j + " ");
        }
    }
```

Поскольку блок, помеченный меткой `one`, не содержит оператор `break`, передача управления этому внешнему блоку невозможна.

Использование оператора `continue`

Иногда требуется, чтобы повторение цикла осуществлялось с более раннего оператора его тела. То есть на данной конкретной итерации может требоваться

продолжить выполнение цикла без обработки остального кода в его теле. По сути, это означает переход в теле цикла к его окончанию. Для выполнения этого действия служит оператор `continue`. В циклах `while` и `do-while` оператор `continue` вызывает передачу управления непосредственно управляющему условному выражению цикла. В цикле `for` управление передается вначале итерационной части цикла `for`, а потом условному выражению. Во всех этих трех циклах любой промежуточный код пропускается.

Ниже приведен пример программы, в которой оператор `continue` используется для вывода двух чисел в каждой строке.

```
// Демонстрация применения оператора 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("");
        }
    }
}
```

В этом коде оператор `%` служит для проверки четности значения переменной `i`. Если оно четное, выполнение цикла продолжается без перехода к новой строке. Программа создает следующий вывод.

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

Как и оператор `break`, оператор `continue` может содержать метку содержащего его цикла, который нужно продолжить. Ниже показан пример программы, в которой оператор `continue` применяется для вывода треугольной таблицы умножения чисел от 0 до 9.

```
// Использование оператора 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();
    }
}
```

В этом примере оператор `continue` прерывает цикл подсчета значений переменной `j` и продолжает его со следующей итерации цикла подсчета переменной `i`. Вывод этой программы имеет следующий вид.

```
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
```

Удачные применения оператора `continue` встречаются редко. Одна из причин состоит в том, что язык Java предлагает широкий выбор операторов цикла, удовлетворяющих требованиям большинства приложений. Однако в тех случаях, когда требуется более раннее начало новой итерации, оператор `continue` предоставляет структурированный метод выполнения этой задачи.

Оператор `return`

Последний из управляющих операторов — `return`. Его используют для выполнения явного выхода из метода. То есть он снова передает управление объекту, который вызвал данный метод. Как таковой, этот оператор относится к операторам перехода. Хотя полное описание оператора `return` придется отложить до рассмотрения методов в главе 6, все же кратко ознакомимся с его особенностями.

Оператор `return` можно использовать в любом месте метода для возврата управления тому объекту, который вызвал данный метод. Таким образом, оператор `return` немедленно прекращает выполнение метода, в котором он находится. Следующий пример иллюстрирует это. В данном случае оператор `return` приводит к возврату управления системе времени выполнения Java, поскольку именно она вызывает метод `main()`.

```
// Демонстрация использования оператора return.
class Return {
    public static void main(String args[]) {
        boolean t = true;

        System.out.println("До выполнения возврата.");

        if(t) return; // возврат к вызывающему объекту

        System.out.println("Этот оператор выполняться не будет.");
    }
}
```

Вывод этой программы имеет такой вид.

```
До выполнения возврата.
```

Как видите, заключительный вызов метода `println()` не выполняется. Сразу после выполнения оператора `return` программа возвращает управление вызывающему объекту.

И последний нюанс: в приведенной программе использование оператора `if(t)` обязательно. Без него компилятор Java сигнализировал бы об ошибке "unreachable code" ("недостижимый код"), поскольку выяснил бы, что последний вызов метода `println()` никогда не будет выполняться. Во избежание этой ошибки в демонстрационном примере пришлось ввести компилятор в заблуждение с помощью оператора `if`.

ГЛАВА 6

Знакомство с классами

Класс — центральный компонент Java. Поскольку класс определяет форму и сущность объекта, он является той логической конструкцией, на основе которой построен весь язык. Как таковой, класс образует основу объектно-ориентированного программирования в среде Java. Любая концепция, которую нужно реализовать в программе Java, должна быть помещена внутрь класса. В связи с тем, что класс имеет такое большое значение для языка Java, эта и несколько следующих глав посвящены классам. В этой главе читатели ознакомятся с основными элементами класса и узнают, как можно использовать класс для создания объектов. Читатели ознакомятся также с методами, конструкторами и ключевым словом `this`.

Основы классов

Мы пользовались классами с самого начала этой книги. Однако до сих пор демонстрировалась только наиболее примитивная форма класса. Классы, созданные в предшествующих главах, служили только в качестве контейнеров метода `main()`, который мы использовали для ознакомления с основами синтаксиса языка Java. Как вы вскоре убедитесь, классы предоставляют значительно больше возможностей, чем те, которые были представлены до сих пор.

Вероятно, наиболее важное свойство класса то, что он определяет новый тип данных. После того как он определен, этот новый тип можно применять для создания объектов данного типа. Таким образом, класс — это *шаблон* объекта, а объект — это *экземпляр* класса. Поскольку объект является экземпляром класса, термины *объект* и *экземпляр* используются как синонимы.

Общая форма класса

При определении класса объявляют его конкретную форму и сущность. Для этого указывают данные, которые он содержит, и кода, воздействующего на эти данные. Хотя очень простые классы могут содержать только код или только данные, большинство классов, применяемых в реальных программах, содержит оба эти компонента. Как будет показано в дальнейшем, код класса определяет интерфейс к его данным.

Для объявления класса служит ключевое слово `class`. Использованные до сих пор классы в действительности представляли собой очень ограниченные примеры полной формы. Классы могут быть (и обычно являются) значительно более сложными. Упрощенная общая форма определения класса имеет следующий вид.

```
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 внутри класса, называются *переменными экземпляра*. Код содержится внутри *методов*. Определенные внутри класса методы и переменные вместе называют *членами* класса. В большинстве классов действия с переменными экземпляров и доступ к ним осуществляют методы, определенные в этом классе. Таким образом, в общем случае именно методы определяют способ использования данных класса.

Определенные внутри класса переменные называют переменными экземпляра, поскольку каждый экземпляр класса (т.е. каждый объект класса) содержит собственные копии этих переменных. Таким образом, данные одного объекта отделены и отличаются от данных другого объекта. Вскоре мы вернемся к рассмотрению этой концепции, но она слишком важна, чтобы можно было обойтись без хотя бы предварительного ознакомления с нею.

Все методы имеют ту же общую форму, что и метод `main()`, который мы использовали до сих пор. Однако большинство методов не будет указано как `static` или `public`. Обратите внимание на то, что общая форма класса не содержит определения метода `main()`. Классы Java могут и не содержать этот метод. Его обязательно указывать только в тех случаях, когда данный класс служит начальной точкой программы. Более того, некоторые типы приложений Java, такие как аплеты, вообще не требуют использования метода `main()`.

На заметку! Программисты на языке C++ обратят внимание на то, что объявление класса и реализация методов хранятся в одном месте, а не определены отдельно. Иногда эта особенность приводит к созданию очень больших файлов `.java`, поскольку любой класс должен быть полностью определен в одном файле исходного кода. Такая архитектура была принята для Java умышленно, поскольку разработчики посчитали, что хранение определения, объявления и реализации в одном файле упрощает сопровождение кода в течение длительного периода его эксплуатации.

Простой класс

Изучение классов начнем с простого примера. Ниже приведен код класса `Box` (Параллелепипед), который определяет три переменные экземпляра: `width` (ширина), `height` (высота) и `depth` (глубина). В настоящий момент класс `Box` не содержит никаких методов (но вскоре мы добавим в него метод).

```
class Box {
    double width;
    double height;
    double depth;
}
```

Как уже было сказано, класс определяет новый тип данных. В данном случае новый тип данных назван Box. Это имя будет использоваться для объявления объектов типа Box. Важно помнить, что объявление class создает только шаблон, но не действительный объект. Таким образом, приведенный код не приводит к появлению никаких объектов типа Box.

Чтобы действительно создать объект класса Box, нужно использовать оператор, подобный следующему.

```
Box mybox = new Box(); // создание объекта mybox класса Box
```

После выполнения этого оператора mybox станет экземпляром класса Box. То есть он обретет "физическое" существование. Но пока можете не задумываться о нюансах этого оператора.

Повторим еще раз: при каждом создании экземпляра класса мы создаем объект, который содержит собственную копию каждой переменной экземпляра, определенной классом. Таким образом, каждый объект класса Box будет содержать собственные копии переменных экземпляра width, height и depth. Для доступа к этим переменным применяется оператор *точки* (.). Этот оператор связывает имя объекта с именем переменной экземпляра. Например, чтобы присвоить переменной width объекта mybox значение 100, нужно было бы использовать следующий оператор.

```
mybox.width = 100;
```

Этот оператор указывает компилятору, что копии переменной width, хранящейся внутри объекта mybox, нужно присвоить значение 100. В общем случае точечный оператор используют для доступа как к переменным экземпляра, так и к методам внутри объекта. Еще один момент: хотя обычно точку (.) называют точечным *оператором*, формальная спецификация языка Java относит его к категории разделителей. Однако поскольку термин "точечный оператор" широко распространен, он и используется в этой книге.

Ниже приведена полная программа, в которой используется класс Box.

```
/* Программа, использующая класс Box.

   Назовите этот файл BoxDemo.java
*/
class Box {
    double width;
    double height;
    double depth;
}

// Этот класс объявляет объект класса Box.
class BoxDemo {
    public static void main(String args[]) {
        Box mybox = new Box();
        double vol;

        // присваивание значений переменным экземпляра mybox
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;

        // вычисление объема параллелепипеда
        vol = mybox.width * mybox.height * mybox.depth;
        System.out.println("Объем равен " + vol);
    }
}
```

Файлу этой программы нужно присвоить имя BoxDemo.java, поскольку метод main() определен в классе, названном BoxDemo, а не Box. Выполнив компиляцию этой программы, вы убедитесь в создании двух файлов .class: для имени Box и для BoxDemo. Компилятор Java автоматически помещает каждый класс в отдельный файл с расширением .class. В действительности классы Box и BoxDemo не обязательно должны быть объявлены в одном и том же исходном файле. Каждый класс можно было бы поместить в отдельный файл, названный соответственно Box.java и BoxDemo.java.

Чтобы запустить эту программу, нужно выполнить файл BoxDemo.class. В результате будет получен следующий вывод.

```
Объем равен 3000.0
```

Как было сказано ранее, каждый объект содержит собственные копии переменных экземпляра. Это означает, что при наличии двух объектов класса Box каждый из них будет содержать собственные копии переменных depth, width и height. Важно понимать, что изменения переменных экземпляра одного объекта не влияют на переменные экземпляра другого. Например, в следующей программе объявлены два объекта класса Box.

```
// Эта программа объявляет два объекта класса 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;

        // присваивание значений переменным экземпляра mybox1
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        //* присваивание других значений переменным
            экземпляра mybox2's */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // вычисление объема первого параллелепипеда
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Объем равен " + vol);

        // вычисление объема второго параллелепипеда
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
    }
}
```

Эта программа создает следующий вывод.

```
Объем равен 3000.0
Объем равен 162.0
```

Как видите, данные объекта mybox1 полностью изолированы от данных, содержащихся в объекте mybox2.

Объявление объектов

Как мы уже отмечали, при создании класса вы создаете новый тип данных. Этот тип можно использовать для объявления объектов данного типа. Однако создание объектов класса – двухступенчатый процесс. Вначале необходимо объявить переменную типа класса. Эта переменная не определяет объект. Она представляет собой всего лишь переменную, которая может *ссылаться* на объект. Затем потребуется получить действительную, физическую копию объекта и присвоить ее этой переменной. Это можно сделать при помощи оператора `new`. Этот оператор динамически (т.е. во время выполнения) резервирует память под объект и возвращает ссылку на него. В общих чертах эта ссылка представляет собой адрес объекта в памяти, зарезервированной оператором `new`. Затем эта ссылка сохраняется в переменной. Таким образом, в Java все объекты классов должны создаваться динамически. Рассмотрим эту процедуру более подробно.

В приведенном ранее примере программы строка, подобная следующей, используется для объявления объекта класса `Box`.

```
Box mybox = new Box();
```

Этот оператор объединяет только что описанные этапы. Чтобы каждый этап был более очевидным, его можно переписать следующим образом.

```
Box mybox; // объявление ссылки на объект
mybox = new Box(); // резервирование памяти для объекта Box
```

Первая строка объявляет `mybox` ссылкой на объект класса `Box`. После выполнения этой строки переменная `mybox` содержит значение `null`, свидетельствующее о том, что она еще не указывает на реальный объект. Любая попытка использования переменной `mybox` на этом этапе приведет к ошибке времени компиляции. Следующая строка резервирует память под реальный объект и присваивает переменной `mybox` ссылку на этот объект. После выполнения второй строки переменную `mybox` можно использовать, как если бы она была объектом класса `Box`. Но в действительности переменная `mybox` просто содержит адрес памяти реального объекта класса `Box`. Результат выполнения этих двух строк кода показан на рис. 6.1.

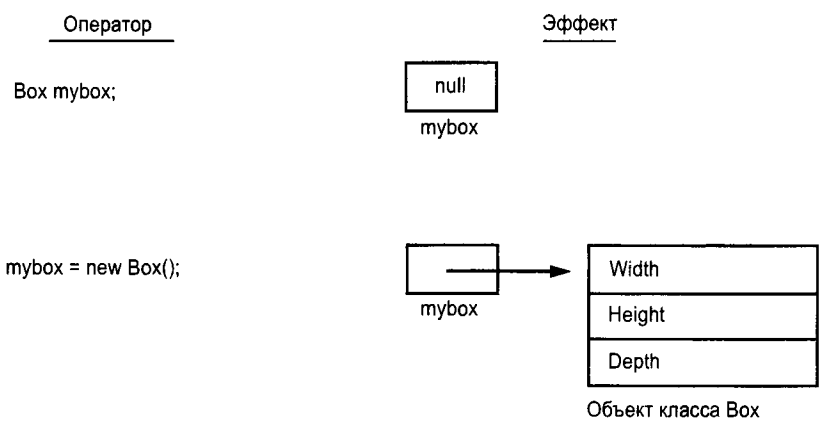


Рис 6.1. Объявление объекта класса `Box`

На заметку! Читатели, которые знакомы с языками C/C++, вероятно, заметили, что ссылки на объекты подобны указателям. В общих чертах это верно. Ссылка на объект похожа на указатель памяти. Основное различие между ними — и основное свойство, обеспечивающее безопасность программ Java, — в том, что ссылками нельзя манипулировать, как настоящими указателями. В частности, ссылка на объект не может указывать на произвольную ячейку памяти, и ею нельзя манипулировать как целочисленным значением.

Подробное рассмотрение оператора new

Как было сказано, оператор new динамически резервирует память для объекта. Общая форма этого оператора имеет следующий вид.

```
переменная_класса = new имя_класса ();
```

Здесь *переменная_класса* — переменная создаваемого класса. *Имя_класса* — это имя класса, конкретизация которого выполняется. Имя класса, за которым следуют круглые скобки, указывает *конструктор* данного класса. Конструктор определяет действия, выполняемые при создании объекта класса. Конструкторы — важная часть всех классов, и они обладают множеством важных атрибутов. Большинство классов, используемых в реальных программах, явно определяют свои конструкторы внутри определения класса. Однако если никакой явный конструктор не указан, Java автоматически предоставит конструктор, используемый по умолчанию. Это же происходит в случае объекта класса Box. Пока мы будем пользоваться конструктором, заданным по умолчанию. Вскоре читатели научатся определять собственные конструкторы.

У читателей может возникнуть вопрос, почему не требуется использовать оператор new для таких элементов, как целые числа или символы. Это обусловлено тем, что элементарные типы Java реализованы в не виде объектов, а в виде "обычных" переменных. Это сделано для повышения эффективности. Как вы убедитесь, объекты обладают множеством свойств и атрибутов, которые требуют, чтобы программа Java обрабатывала их иначе, чем элементарные типы. Отсутствие накладных расходов, связанных с обработкой объектов, при обработке элементарных типов позволяет эффективнее реализовать элементарные типы. Несколько позже мы приведем объектные версии элементарных типов, которые могут пригодиться в ситуациях, когда требуются полноценные объекты этих типов.

Важно понимать, что оператор new резервирует память для объекта во время выполнения. Преимущество этого подхода состоит в том, что программа может создавать ровно столько объектов, сколько требуется во время ее выполнения. Однако поскольку объем памяти ограничен, возможна ситуация, когда оператор new не сможет выделить память для объекта из-за ее нехватки. В этом случае передается исключение времени выполнения. (Обработка исключений описана в главе 10.) В примерах программ, приведенных в этой книге, можно не беспокоиться по поводу недостатка объема памяти, но в реальных программах эту возможность придется учитывать.

Еще раз рассмотрим различие между классом и объектом. Класс создает новый тип данных, который можно использовать для создания объектов. То есть класс создает логический каркас, определяющий взаимосвязь между его членами. При объявлении объекта класса мы создаем экземпляр этого класса. Таким образом, класс — это логическая конструкция. А объект обладает физической сущностью. (То есть объект занимает область в памяти.) Важно помнить об этом различии.

Присваивание переменных объектных ссылок

При выполнении присваивания переменные объектных ссылок действуют иначе, чем можно было бы представить. Например, какие действия, по вашему мнению, выполняет следующий фрагмент кода?

```
Box b1 = new Box();
Box b2 = b1;
```

Можно подумать, что переменной b2 присваивается ссылка на копию объекта, на которую ссылается переменная b1. То есть может показаться, что переменные b1 и b2 ссылаются на отдельные и различные объекты. Однако это не так. После выполнения этого фрагмента кода обе переменные b1 и b2 будут ссылаться на *один и тот же* объект. Присваивание переменной b1 переменной b2 не привело к резервированию какой-то области памяти или копированию какой-либо части исходного объекта. Этот оператор присваивания приводит лишь к тому, что переменная b2 ссылается на тот же объект, что и переменная b1. Таким образом, любые изменения, выполненные в объекте через переменную b2, окажут влияние на объект, на который ссылается переменная b1, поскольку это — один и тот же объект.

Эта ситуация отражена на рис. 6.2.

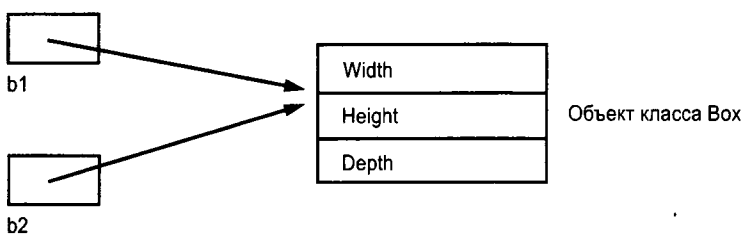


Рис. 6.2. Использование переменных объектных ссылок

Хотя и переменные b1 и b2 ссылаются на один и тот же объект, они не связаны между собой никаким другим образом. Например, следующий оператор присваивания значения переменной b1 просто *разорвет связь* переменной b1 с исходным объектом, не оказывая влияния на сам объект или переменную b2.

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

В этом примере значение переменной b1 установлено равным null, но переменная b2 по-прежнему указывает на исходный объект.

Помните! Присваивание ссылочной переменной одного объекта ссылочной переменной другого объекта не ведет к созданию копии объекта, а лишь создает копию ссылки.

Знакомство с методами

Как было сказано в начале этой главы, обычно классы состоят из двух элементов: переменных экземпляра и методов. Поскольку язык Java предоставляет им столь большие возможности и гибкость, тема методов очень обширна. Фактически многие последующие главы посвящены методам. Однако чтобы можно было при-

ступить к добавлению методов к своим классам, необходимо ознакомиться с рядом их основных характеристик.

Общая форма объявления метода выглядит следующим образом.

```
тип имя(список_параметров) {
    // тело метода
}
```

Здесь *тип* указывает тип данных, возвращаемых методом. Он может быть любым допустимым типом, в том числе типом класса, созданным программистом. Если метод не возвращает значение, типом его возвращаемого значения должен быть `void`. *Имя* служит для указания имени метода. Оно может быть любым допустимым идентификатором, кроме тех, которые уже используются другими элементами в текущей области видимости. *Список_параметров* — последовательность пар "тип-идентификатор", разделенных запятыми. По сути, параметры — это переменные, которые принимают значения аргументов, переданных методу во время его вызова. Если метод не имеет параметров, список параметров будет пустым.

Методы, тип возвращаемого значения которых отличается от `void`, возвращают значение вызывающей процедуре с помощью следующей формы оператора `return`.

```
return значение;
```

Здесь *значение* — это возвращаемое значение.

В последующих разделах мы рассмотрим создание различных типов методов, включая принимающие параметры и возвращающие значения.

Добавление метода к классу Box

Хотя было бы весьма удобно создать класс, который содержит только данные, в реальных программах подобное встречается редко. В большинстве случаев для доступа к переменным экземпляра, определенным классом, придется использовать методы. Фактически методы определяют интерфейсы большинства классов. Это позволяет программисту, который реализует класс, скрывать конкретную схему внутренних структур данных за более понятными абстракциями метода. Кроме определения методов, которые обеспечивают доступ к данным, можно определять также методы, используемые внутренне самим классом.

Теперь приступим к добавлению метода в класс `Box`. Просматривая предшествующие программы, легко прийти к выводу, что класс `Box` мог бы лучше справиться с вычислением объема параллелепипеда, чем класс `BoxDemo`. В конце концов, поскольку объем параллелепипеда зависит от его размеров, вполне логично, чтобы его вычисление выполнялось в классе `Box`. Для этого в класс `Box` нужно добавить метод, как показано в следующем примере.

```
// Эта программа содержит метод внутри класса box.

class Box {
    double width;
    double height;
    double depth;

    // отображение объема параллелепипеда
    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();

        // присваивание значений переменным экземпляра mybox1
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* присваивание других значений переменным
           экземпляра mybox2 */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // отображение объема первого параллелепипеда
        mybox1.volume();

        // отображение объема второго параллелепипеда
        mybox2.volume();
    }
}
```

Эта программа создает следующий вывод, совпадающий с выводом предыдущей версии.

```
Объем равен 3000.0
Объем равен 162.0
```

Внимательно взгляните на следующие две строки кода.

```
mybox1.volume();
mybox2.volume();
```

В первой строке присутствует обращение к методу `volume()` объекта `mybox1`. То есть она вызывает метод `volume()` по отношению к объекту `mybox1`, для чего было использовано имя объекта, за которым следует точечный оператор. Таким образом, обращение к методу `mybox1.volume()` отображает объем параллелепипеда, определенного объектом `mybox1`, а обращение к методу `mybox2.volume()` — объем параллелепипеда, определенного объектом `mybox2`. При каждом вызове метод `volume()` отображает объем указанного параллелепипеда.

Соображения, приведенные в следующих абзацах, облегчат понимание концепции вызова метода. При вызове метода `mybox1.volume()` система времени выполнения Java передает управление коду, определенному внутри метода `volume()`. По завершении выполнения всех операторов внутри метода управление возвращается вызывающей программе, и ее выполнение продолжается со строки, которая следует за вызовом метода. В самом общем смысле можно сказать, что метод – способ реализации подпрограмм в Java.

В методе `volume()` следует обратить внимание на один очень важный нюанс: ссылка на переменные экземпляра `width`, `height` и `depth` выполняется непосредственно, без указания перед ними имени объекта или точечного оператора. Когда метод использует переменную экземпляра, которая определена его классом, он выполняет это непосредственно, без указания явной ссылки на объект и без применения точечного оператора. Это становится понятным, если немного подумать. Метод всегда вызывается по отношению к какому-то объекту его класса. Как только этот вызов выполнен, объект известен. Таким образом, внутри метода вторичное указание объекта совершенно излишне. Это означает, что переменные

width, height и depth неявно ссылаются на копии этих переменных, хранящиеся в объекте, который вызывает метод volume().

Подведем краткие итоги. Когда обращение к переменной экземпляра выполняется кодом, не являющимся частью класса, в котором определена переменная экземпляра, необходимо указать объект при помощи точечного оператора. Однако когда это обращение осуществляется кодом, который является частью того же класса, где определена переменная экземпляра, ссылка на переменную может выполняться непосредственно. Эти же правила применимы и к методам.

Возвращение значения

Хотя реализация метода volume() переносит вычисление объема параллелепипеда внутрь класса Box, которому принадлежит этот метод, такой способ вычисления не является наилучшим. Например, что делать, если другой части программы требуется знание объема параллелепипеда без его отображения? Более рациональный способ реализации метода volume() — вычисление объема параллелепипеда и возврат результата вызывающему объекту. Следующий пример — усовершенствованная версия предыдущей программы — выполняет именно эту задачу.

```
// Теперь метод volume() возвращает объем параллелепипеда.

class Box {
    double width;
    double height;
    double depth;

    // вычисление и возвращение объема
    double volume() {
        return width * height * depth;
    }
}
class BoxDemo4 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // присваивание значений переменным экземпляра mybox1
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* присваивание других значений переменным экземпляра mybox2 */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // получение объема первого параллелепипеда
        vol = mybox1.volume();
        System.out.println("Объем равен " + vol);

        // получение объема второго параллелепипеда
        vol = mybox2.volume();
        System.out.println("Объем равен " + vol);
    }
}
```

Как видите, вызов метода `volume()` выполняется в правой части оператора присваивания. Правой частью этого оператора является переменная, в данном случае `vol`, которая будет принимать значение, возвращенное методом `volume()`. Таким образом, после выполнения оператора

```
vol = mybox1.volume();
```

метод `mybox1.volume()` возвратит значение 3000, и этот объем сохраняется в переменной `vol`. При работе с возвращаемыми значениями следует учитывать два важных обстоятельства.

- Тип данных, возвращаемых методом, должен быть совместим с возвращаемым типом, указанным методом. Например, если возвращаемым типом какого-либо метода является `boolean`, нельзя возвращать целочисленное значение.
- Переменная, принимающая возвращенное методом значение (такая, как `vol`), также должна быть совместима с возвращаемым типом, указанным для метода.

И еще один нюанс: предыдущую программу можно было бы записать в несколько более эффективной форме, поскольку в действительности переменная `vol` совершенно не нужна. Обращение к методу `volume()` можно было бы использовать в вызове метода `println()` непосредственно, как в следующей строке кода.

```
System.out.println("Объем равен" + mybox1.volume());
```

В этом случае при вызове метода `println()` метод `mybox1.volume()` будет вызываться автоматически, а возвращаемое им значение будет передаваться методу `println()`.

Добавление метода, принимающего параметры

Хотя некоторые методы не нуждаются в параметрах, большинство требует их передачи. Параметры позволяют обобщать метод. То есть метод с параметрами может работать с различными данными и/или применяться в ряде несколько различных ситуаций. В качестве иллюстрации рассмотрим очень простой пример. Ниже показан метод, который возвращает квадрат числа 10.

```
int square()
{
    return 10 * 10;
}
```

Хотя этот метод действительно возвращает 102, его применение очень ограничено. Однако если его изменить так, чтобы он принимал параметр, как показано в следующем примере, метод `square()` может стать значительно более полезным.

```
int square(int i)
{
   return i * i;
}
```

Теперь метод `square()` будет возвращать квадрат любого значения, с которым он вызван. То есть теперь метод `square()` является методом общего назначения, который может вычислять квадрат любого целочисленного значения, а не только числа 10.

Приведем примеры.

```
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 обращении к методу `square()` значение 5 будет передано параметру `i`. Во втором обращении параметр `i` примет значение, равное 9. Третий вызов метода передает значение переменной `y`, которое в этом примере составляет 2. Как видно из этих примеров, метод `square()` способен возвращать квадрат любых переданных ему данных.

Важно различать два термина: *параметр* и *аргумент*. *Параметр* — это определенная в методе переменная, которая принимает значение при вызове метода. Например, в методе `square()` параметром является `i`. *Аргумент* — это значение, передаваемое методу при его вызове. Например, методу `square(100)` в качестве аргумента передается значение 100. Внутри метода `square()` параметр `i` получает это значение.

Метод с параметрами можно использовать для усовершенствования класса `Box`. В предшествующих примерах размеры каждого параллелепипеда нужно было устанавливать отдельно, используя последовательность операторов вроде следующей.

```
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

Хотя этот код работает, он не очень удобен по двум причинам. Во-первых, он громоздкий и чреват ошибками. Например, вполне можно забыть определить один из размеров. Во-вторых, в правильно спроектированных программах Java доступ к переменным экземпляра должен осуществляться только через методы, определенные их классом. В будущем поведение метода можно изменить, но нельзя изменить поведение предоставленной переменной экземпляра.

Поэтому более рациональный способ установки размеров параллелепипеда — создание метода, который принимает размеры параллелепипеда в виде своих параметров и соответствующим образом устанавливает значение каждой переменной экземпляра. Эта концепция реализована в приведенной ниже программе.

```
// Эта программа использует метод с параметрами.

class Box {
    double width;
    double height;
    double depth;

    // вычисление и возвращение объема
    double volume() {
        return width * height * depth;
    }
    // установка размеров параллелепипеда
    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;
```

```
        // инициализация каждого экземпляра Box
        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);

        // получение объема первого параллелепипеда
        vol = mybox1.volume();
        System.out.println("Объем равен " + vol);

        // получение объема второго параллелепипеда
        vol = mybox2.volume();
        System.out.println("Объем равен " + vol);
    }
}
```

Как видите, метод `setDim()` использован для установки размеров каждого параллелепипеда. Например, при выполнении оператора

```
mybox1.setDim(10, 20, 15);
```

значение 10 копируется в параметр `w`, 20 — в `h` и 15 — в `d`. Затем внутри метода `setDim()` значения параметров `w`, `h` и `d` присваиваются соответственно переменным `width`, `height` и `depth`.

Концепции, представленные в этих разделах, вероятно, знакомы многим читателям. Но если вы еще не знакомы с такими понятиями, как вызовы методов, аргументы и параметры, можете немного поэкспериментировать с ними, прежде чем продолжить изучение материала, изложенного в последующих разделах. Концепции вызова метода, параметров и возвращаемых значений являются основополагающими в программировании на языке Java.

Конструкторы

Инициализация всех переменных класса при каждом создании его экземпляра может оказаться утомительным процессом. Даже при добавлении функций, предназначенных для увеличения удобства работы, таких как метод `setDim()`, было бы проще и удобнее, если бы все действия по установке значений переменных выполнялись при первом создании объекта. Поскольку необходимость инициализации возникает столь часто, язык Java позволяет объектам выполнять собственную инициализацию при их создании. Эта автоматическая инициализация осуществляется с помощью конструктора.

Конструктор инициализирует объект непосредственно во время создания. Его имя совпадает с именем класса, в котором он находится, а синтаксис аналогичен синтаксису метода. Как только он определен, конструктор автоматически вызывается непосредственно после создания объекта, перед завершением выполнения оператора `new`. Конструкторы выглядят несколько непривычно, поскольку не имеют типа возвращаемого значения, даже типа `void`. Это обусловлено тем, что неявно заданным возвращаемым типом конструктора класса является тип самого класса. Именно конструктор инициализирует внутреннее состояние объекта так, чтобы код, создающий экземпляр, с самого начала содержал полностью инициализированный, пригодный к использованию объект.

Пример класса `Box` можно изменить, чтобы значения размеров параллелепипеда присваивались при конструировании объекта. Для этого потребуется заменить метод `setDim()` конструктором. Вначале определим простой конструктор, который просто устанавливает одинаковые значения размеров для всех параллелепипедов. Эта версия программы имеет такой вид.

```
/* В этом примере класс Box использует конструктор
   для инициализации размеров параллелепипеда.
*/
class Box {
    double width;
    double height;
    double depth;

    // Это конструктор класса Box.
    Box() {
        System.out.println("Конструирование объекта Box");
        width = 10;
        height = 10;
        depth = 10;
    }

    // вычисление и возвращение объема
    double volume() {
        return width * height * depth;
    }
}
class BoxDemo6 {
    public static void main(String args[]) {
        // объявление, резервирование и инициализация объектов Box
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        double vol;

        // получение объема первого параллелепипеда
        vol = mybox1.volume();
        System.out.println("Объем равен " + vol);

        // получение объема второго параллелепипеда
        vol = mybox2.volume();
        System.out.println("Объем равен " + vol);
    }
}
```

Эта программа создает следующий вывод.

```
Конструирование объекта Box
Конструирование объекта Box
Объем равен 1000.0
Объем равен 1000.0
```

Как видите, и объект `mybox1`, и объект `mybox2` были инициализированы конструктором `Box()` при их создании. Поскольку конструктор присваивает всем параллелепипедам одинаковые размеры 10×10×10, объекты `mybox1` и `mybox2` будут иметь одинаковый объем. Вызов метода `println()` внутри конструктора `Box()` служит исключительно иллюстративным целям.

Большинство конструкторов не выводят никакой информации, а лишь выполняют инициализацию объекта.

Прежде чем продолжить, еще раз рассмотрим оператор `new`. Как вы уже знаете, при резервировании памяти для объекта используют следующую общую форму.

```
переменная_класса = new имя_класса();
```

Теперь вам должно быть ясно, почему после имени класса требуются круглые скобки. В действительности этот оператор вызывает конструктор класса. Таким образом, в строке.

```
Box mybox1 = new Box();
```

оператор `new Box()` вызывает конструктор `Box()`. Если конструктор класса не определен явно, Java создает для класса конструктор, который будет использоваться по умолчанию. Именно поэтому приведенная строка кода работала в предыдущих версиях класса `Box`, в которых конструктор не был определен. Конструктор, используемый по умолчанию, инициализирует все переменные экземпляра нулевыми значениями. Зачастую конструктора, используемого по умолчанию, вполне достаточно для простых классов, чего обычно нельзя сказать о более сложных. Как только конструктор определен, конструктор, заданный по умолчанию, больше не используется.

Конструкторы с параметрами

Хотя в предыдущем примере конструктор `Box()` инициализирует объект класса `Box`, он не особенно полезен — все параллелепипеды получают одинаковые размеры. Следовательно, необходим способ создания объектов класса `Box` с различными размерами. Простейшее решение этой задачи — добавление к конструктору параметров. Как легко догадаться, это делает конструктор значительно более полезным. Например, следующая версия класса `Box` определяет конструктор с параметрами, который устанавливает размеры параллелепипеда в соответствии со значениями этих параметров. Обратите особое внимание на способ создания объектов класса `Box`.

```
/* В этой программе класс Box использует конструктор с параметрами
   для инициализации размеров параллелепипеда.
*/
class Box {
    double width;
    double height;
    double depth;

    // Это конструктор класса Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // вычисление и возврат объема
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo7 {
    public static void main(String args[]) {
        // объявление, резервирование и инициализация объектов Box
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);

        double vol;

        // получение объема первого параллелепипеда
        vol = mybox1.volume();
        System.out.println("Объем равен " + vol);

        // получение объема второго параллелепипеда
        vol = mybox2.volume();
        System.out.println("Объем равен " + vol);
    }
}
```

Вывод этой программы имеет следующий вид.

```
Объем равен 3000.0
Объем равен 162.0
```

Как видите, инициализация каждого объекта выполняется в соответствии со значениями, указанными в параметрах его конструктора. Например, в строке

```
Box mybox1 = new Box(10, 20, 15);
```

значения 10, 20 и 15 передаются конструктору `Box()` при создании объекта с использованием оператора `new`. Таким образом, копии переменных `width`, `height` и `depth` будут содержать соответственно значения 10, 20 и 15.

Ключевое слово `this`

Иногда необходимо, чтобы метод ссылался на вызвавший его объект. Чтобы это было возможно, в Java определено ключевое слово `this`. Оно может использоваться внутри любого метода для ссылки на текущий объект. То есть `this` всегда служит ссылкой на объект, для которого был вызван метод. Ключевое слово `this` можно использовать везде, где допускается ссылка на объект типа текущего класса.

Для пояснения рассмотрим следующую версию конструктора `Box()`.

```
// Избыточное применение ключевого слова this.
Box(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this.depth = d;
}
```

Эта версия конструктора `Box()` действует точно так же, как предыдущая. Применение ключевого слова `this` избыточно, но совершенно правильно. Внутри метода `Box()` ключевое слово `this` всегда будет ссылаться на вызывающий объект. Хотя в данном случае это и излишне, в других случаях, один из которых рассмотрен в следующем разделе, ключевое слово `this` весьма полезно.

Сокрытие переменной экземпляра

Как вы знаете, в языке Java не допускается объявление двух локальных переменных с одним и тем же именем в одной и той же или во включающих одна другую областях видимости. Интересно отметить, что могут существовать локальные переменные, в том числе формальные параметры методов, которые совпадает с именами переменных экземпляра класса. Однако когда имя локальной переменной совпадает с именем переменной экземпляра, локальная переменная *скрывает* переменную экземпляра. Именно поэтому внутри класса `Box` переменные `width`, `height` и `depth` не были использованы в качестве имен параметров конструктора `Box()`. В противном случае переменная `width`, например, ссылалась бы на формальный параметр, скрывая переменную экземпляра `width`. Хотя обычно проще использовать различные имена, существует и другой способ выхода из подобной ситуации. Поскольку ключевое слово `this` позволяет ссылаться непосредственно на объект, его можно применять для разрешения любых конфликтов пространства имен, которые могут возникать между переменными экземпляра и локальными переменными. Например, ниже показана еще одна версия метода `Box()`, в которой имена `width`, `height` и `depth` использованы в качестве имен параметров,

а ключевое слово this служит для обращения к переменным экземпляра по этим же именам.

```
// Этот код служит для разрешения конфликтов пространства имен.
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

Небольшое предостережение: иногда подобное применение ключевого слова this может приводить к недоразумениям, и некоторые программисты стараются не применять имена локальных переменных и параметров, скрывающие переменные экземпляров. Конечно, многие из программистов придерживаются иного мнения и считают целесообразным для облегчения понимания программ использовать одни и те же имена, а для предотвращения сокрытия переменных экземпляров применяют ключевое слово this. Выбор того или иного подхода — дело личного вкуса.

Сбор "мусора"

Поскольку резервирование памяти для объектов осуществляется динамически при помощи оператора new, у читателей может возникнуть вопрос, как уничтожаются такие объекты и как их память освобождается для последующего резервирования. В некоторых языках, подобных C++, динамически резервированные объекты нужно освобождать вручную с помощью оператора delete. В языке Java применяется другой подход. Освобождение памяти выполняется автоматически. Используемая для выполнения этой задачи технология называется *сбором "мусора"*. Процесс проходит следующим образом: при отсутствии каких-либо ссылок на объект программа заключает, что этот объект больше не нужен и занимаемую объектом память можно освободить. В языке Java не нужно явно уничтожать объекты, как это делается в языке C++. Во время выполнения программы сбор "мусора" выполняется только изредка (если вообще выполняется). Она не будет выполняться лишь потому, что один или несколько объектов существуют и больше не используются. Более того, в различных реализациях системы времени выполнения Java могут применяться различные подходы к сбору "мусора", но в большинстве случаев при написании программ об этом можно не беспокоиться.

Метод finalize()

Иногда при уничтожении объект должен выполнять некое действие. Например, если объект содержит какой-то ресурс, отличный от ресурса Java (вроде файлового дескриптора или шрифта), может потребоваться гарантия освобождения этих ресурсов перед уничтожением объекта. Для отработки подобных ситуаций язык Java предоставляет механизм, называемый *финализацией*. Используя финализацию, можно определить конкретные действия, которые будут выполняться непосредственно перед удалением объекта сборщиком "мусора".

Чтобы добавить в класс средство финализации, достаточно определить метод finalize(). Среда времени выполнения Java вызывает этот метод непосредственно перед удалением объекта данного класса. Внутри метода finalize() нужно указать те действия, которые должны быть выполнены перед уничтожени-

ем объекта. Сборщик "мусора" запускается периодически, проверяя наличие объектов, на которые отсутствуют как ссылки со стороны какого-либо текущего состояния, так и косвенные ссылки через другие ссылочные объекты. Непосредственно перед освобождением ресурсов среда времени выполнения Java вызывает метод `finalize()` по отношению к объекту.

Общая форма метода `finalize()` имеет следующий вид.

```
protected void finalize( ){
    // здесь должен находиться код финализации
}
```

В этой синтаксической конструкции ключевое слово `protected` — это модификатор, который предотвращает доступ к методу `finalize()` со стороны кода, определенного вне его класса. Этот и другие модификаторы доступа описаны в главе 7.

Важно понимать, что метод `finalize()` вызывается только непосредственно перед сбором "мусора". Например, он не вызывается при выходе объекта из области видимости. Это означает, что неизвестно, когда будет (и будет ли вообще) выполняться метод `finalize()`. Поэтому программа должна предоставлять другие средства освобождения используемых объектом системных ресурсов и т.п. Нормальная работа программы не должна зависеть от метода `finalize()`.

На заметку! Те читатели, которые знакомы с языком C++, знают, что он позволяет определять деструктор класса, который вызывается при выходе объекта из области видимости. Язык Java не поддерживает эту концепцию и не допускает использование деструкторов. По своему действию метод `finalaize()` лишь отдаленно напоминает деструктор. По мере приобретения опыта программирования на языке Java вы убедитесь, что благодаря наличию подсистемы сбора "мусора" потребность в функциях деструктора очень незначительна.

Класс `Stack`

Хотя класс `Box` удобен для иллюстрации основных элементов класса, его практическая ценность невелика. Чтобы читатели могли убедиться в реальных возможностях классов, изложение материала этой главы мы завершим более сложным примером. Как вы, возможно, помните из материала по основам объектно-ориентированного программирования (ООП), представленного в главе 2, одно из наибольших преимуществ ООП — это инкапсуляция данных и кода, который манипулирует этими данными. Как было показано, в языке Java класс — это механизм инкапсуляции. Создавая класс, вы создаете новый тип данных, который определяет как сущность данных, подлежащих обработке, так и используемые для этого процедуры. Далее методы задают целостный и управляемый интерфейс к данным класса. Таким образом, класс можно использовать за счет его методов, не заботясь о нюансах его реализации или о действительном способе управления данными внутри класса. В определенном смысле класс подобен "машине данных". Чтобы использовать машину при помощи ее элементов управления, не требуются никакие знания о происходящем внутри нее. Фактически, поскольку подробности реализации скрыты, внутренние детали можно изменять по мере необходимости. До тех пор, пока код использует класс через его методы, внутренние детали могут меняться, не вызывая побочных эффектов за пределами класса.

В качестве иллюстрации приведенных соображений рассмотрим один из типичных примеров инкапсуляции — стек. Стек хранит данные в порядке "первым вошел, последним вышел". То есть стек подобен стопке тарелок на столе — тарелка, которая была поставлена на стол первой, будет использована последней. Стеки

управляются двумя операциями, которые традиционно называются заталкиванием (в стек) и выталкиванием (из стека). Для помещения элемента на верхушку стека используется операция заталкивания. Для извлечения элемента из стека применяется операция выталкивания. Как вы убедитесь, инкапсуляция всего механизма стека не представляет сложности.

Ниже приведен код класса, названного `Stack`, который реализует стек размером до десяти целочисленных значений.

```
// Этот класс определяет целочисленный стек, который может
// хранить 10 значений
class Stack {
    int stck[] = new int[10];
    int tos;

    // Инициализация верхушки стека
    Stack() {
        tos = -1;
    }

    // Заталкивание элемента в стек
    void push(int item) {
    if(tos==9)
        System.out.println("Стек полон.");
    else
        stck[++tos] = item;
    }

    // Выталкивание элемента из стека
    int pop() {
        if(tos < 0) {
            System.out.println("Стек не загружен.");
            return 0;
        }
        else
            return stck[tos--];
    }
}
```

Как видите, класс `Stack` определяет два элемента данных и три метода. Стек целочисленных значений хранится в массиве `stck`. Этот массив индексируется по переменной `tos`, которая всегда содержит индекс верхушки стека. Конструктор `Stack()` инициализирует переменную `tos` значением `-1`, которое указывает на пустой стек. Метод `push()` помещает элемент в стек. Чтобы извлечь элемент, нужно вызвать метод `pop()`. Поскольку доступ к стеку осуществляется с использованием методов `push()` и `pop()`, в действительности при работе со стеком не имеет значения, что стек хранится в массиве. Например, стек мог бы храниться в более сложной структуре данных вроде связного списка, но интерфейс, определенный методами `push()` и `pop()`, оставался бы неизменным.

Приведенный в следующем примере класс `TestStack` демонстрирует применение класса `Stack`. Он создает два целочисленных стека, заталкивает в каждый из них определенные значения, а затем выталкивает их из стека.

```
class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        // заталкивает числа в стек
        for(int i=0; i<10; i++) mystack1.push(i);
```

```
        for(int i=10; i<20; i++) mystack2.push(i);

        // выталкивает эти числа из стека
        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());
    }
}
```

Эта программа создает следующий 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
```

Как видите, содержимое обоих стеков различается.

И последнее замечание по поводу класса `Stack`. В том виде, каком он реализован, массив `stck`, который содержит стек, может быть изменен кодом, определенным вне класса `Stack`. Это делает класс `Stack` уязвимым для злоупотреблений и повреждений. В следующей главе будет показано, как можно исправить эту ситуацию.

ГЛАВА 7

Более пристальный взгляд на методы и классы

В этой главе продолжим рассмотрение методов и классов, начатое в предыдущей главе. Вначале рассмотрим несколько вопросов, связанных с методами, в том числе перегрузку, передачу параметров и рекурсию. Затем снова обратимся к классам и рассмотрим управление доступом, использование ключевого слова `static` и один из наиболее важных встроенных классов Java – `String`.

Перегрузка методов

Язык Java разрешает определение внутри одного класса двух или более методов с одним именем, если только объявления их параметров различны. В этом случае методы называют *перегруженными*, а процесс – *перегрузкой методов*. Перегрузка методов – один из способов поддержки полиморфизма в Java. Тем читателям, которые никогда не использовали язык, допускающий перегрузку методов, эта концепция вначале может показаться странной. Но, как вы вскоре убедитесь, перегрузка методов – одна из наиболее впечатляющих и полезных функциональных возможностей языка Java.

При вызове перегруженного метода для определения нужной версии Java использует тип и/или количество аргументов метода. Следовательно, перегруженные методы должны различаться по типу и/или количеству их параметров. Хотя возвращаемые типы перегруженных методов могут быть различны, самого возвращаемого типа не достаточно для различения двух версий метода. Когда среда исполнения Java встречает вызов перегруженного метода, она просто выполняет ту его версию, параметры которой соответствуют аргументам, использованным в вызове.

Следующий простой пример иллюстрирует перегрузку метода.

```
// Демонстрация перегрузки методов.
class OverloadDemo {
    void test() {
        System.out.println("Параметры отсутствуют");
    }

    // Проверка перегрузки на наличие одного целочисленного параметра.
    void test(int a) {
        System.out.println("a: " + a);
    }

    // Проверка перегрузки на наличие двух целочисленных параметров.
    void test(int a, int b) {
        System.out.println("a и b: " + a + " " + b);
    }
```

```
    // Проверка перегрузки на наличие 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;

        // вызов всех версий метода test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Результат ob.test(123.25): " + result);
    }
}
```

Эта программа создает следующий вывод.

```
Параметры отсутствуют
a: 10
a и b: 10 20
double a: 123.25
Результат ob.test(123.25): 15190.5625
```

Как видите, метод `test()` перегружается четыре раза. Первая версия не принимает никаких параметров, вторая принимает один целочисленный параметр, третья — два целочисленных параметра, а четвертая — один параметр типа `double`. То, что четвертая версия метода `test()` возвращает также значение, не имеет никакого значения для перегрузки, поскольку возвращаемый тип никак не влияет на поиск перегруженной версии метода.

При вызове перегруженного метода Java ищет соответствие между аргументами, которые были использованы для вызова метода, и параметрами метода. Однако это соответствие не обязательно должно быть полным. В некоторых случаях при поиске перегруженных версий может использоваться автоматическое преобразование типов Java. Например, рассмотрим следующую программу.

```
// Применение автоматического преобразования типов к перегрузке.
class OverloadDemo {
    void test() {
        System.out.println("Параметры отсутствуют");
    }

    // Проверка перегрузки на наличие двух целочисленных параметров.
    void test(int a, int b) {
        System.out.println("a и b: " + a + " " + b);
    }

    // Проверка перегрузки на наличие параметра типа double
    void test(double a) {
    System.out.println("Внутреннее преобразование 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);     // этот оператор вызовет test(double)
        ob.test(123.2); // этот оператор вызовет test(double)
    }
}
```

Программа создает следующий вывод.

```
Параметры отсутствуют
a и b: 10 20
Внутреннее преобразование test(double) a: 88
Внутреннее преобразование test(double) a: 123.2
```

Как видите, эта версия класса `OverloadDemo` не определяет перегрузку метода `test(int)`. Поэтому при вызове метода `test()` с целочисленным аргументом внутри класса `Overload` соответствующий метод отсутствует. Однако Java может автоматически преобразовать тип `integer` в тип `double`, и это преобразование может использоваться для разрешения вызова. Поэтому после того, как версия метода `test(int)` не обнаружена, Java повышает тип переменной `i` до `double`, а затем вызывает метод `test(double)`. Конечно, если бы метод `test(int)` был определен, то был бы вызван именно он. Java будет использовать автоматическое преобразование типов только при отсутствии полного соответствия.

Перегрузка методов обеспечивает полиморфизм, поскольку это один из способов реализации в Java концепции "один интерфейс, несколько методов". Для пояснения приведем следующие рассуждения. В тех языках, которые не поддерживают перегрузку методов, каждому методу должно быть присвоено уникальное имя. Однако зачастую желательно реализовать, по сути, один и тот же метод для различных типов данных. Например, рассмотрим функцию вычисления абсолютного значения. Обычно в языках, которые не поддерживают перегрузку, существует три или более версий этой функции со слегка различающимися именами. Например, в языке C функция `abs()` возвращает абсолютное значение типа `integer`, функция `labs()` — значения типа `long integer`, а функция `fabs()` — значения с плавающей точкой. Поскольку язык C не поддерживает перегрузку, каждая функция должна обладать собственным именем, несмотря на то что все три функции выполняют по существу одно и то же действие. В результате в концептуальном смысле ситуация становится более сложной, чем она есть на самом деле. Хотя каждая функция построена на основе одной и той же концепции, программист вынужден помнить три имени. В Java подобная ситуация не возникает, поскольку все методы вычисления абсолютного значения могут использовать одно и то же имя. И действительно, стандартная библиотека классов Java содержит метод вычисления абсолютного значения, названный `abs()`. Перегруженные версии этого метода для обработки всех числовых типов определены в классе Java `Math`. Java выбирает для вызова нужную версию метода `abs()` в зависимости от типа аргумента.

Перегрузка методов ценна тем, что позволяет обращаться к схожим методам по общему имени. Таким образом, имя `abs` представляет *общее действие*, которое должно выполняться. Выбор *конкретной* версии для данной ситуации — задача компилятора. Программисту нужно помнить только об общем выполняемом действии. Полиморфизм позволяет свести несколько имен к одному. Приведенный пример весьма прост, но если эту концепцию расширить, легко убедиться в том, что перегрузка может облегчить выполнение более сложных задач.

При перегрузке метода каждая его версия может выполнять любые необходимые действия. Не существует никакого правила, в соответствии с которым перегруженные методы должны быть связаны между собой. Однако со стилистической

точки зрения перегрузка методов предполагает определенную связь. Таким образом, хотя одно и то же имя можно использовать для перегрузки несвязанных методов, поступать так не следует. Например, имя `sqr` можно было бы использовать для создания методов, которые возвращают *квадрат* целочисленного значения и *квадратный корень* значения с плавающей точкой. Но эти две операции принципиально различны. Такое применение перегрузки методов противоречит ее исходному назначению. В частности, следует перегружать только тесно связанные операции.

Перегрузка конструкторов

Наряду с перегрузкой обычных методов можно также выполнять перегрузку методов конструкторов. Фактически перегруженные конструкторы станут нормой, а не исключением, для большинства классов, которые вам придется создавать для реальных программ. Чтобы это утверждение было понятным, вернемся к классу `Box`, разработанному в предыдущей главе. Его последняя версия имеет следующий вид:

```
class Box {
    double width;
    double height;
    double depth;

    // Это конструктор класса Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // вычисление и возврат значения
    double volume() {
        return width * height * depth;
    }
}
```

Как видите, конструктор `Box()` требует передачи трех параметров. Это означает, что все объявления объектов класса `Box` должны передавать конструктору `Box()` три аргумента. Например, следующий оператор недопустим.

```
Box ob = new Box();
```

Поскольку конструктор `Box()` требует передачи трех аргументов, его вызов без аргументов — ошибка. Эта ситуация порождает три важных вопроса. Что если нужно было просто определить параллелепипед и его начальные размеры не имели значения (или не были известны)? Или нужно иметь возможность инициализировать куб, указывая только один размер, который должен использоваться для всех трех измерений? При текущем определении класса `Box` все эти дополнительные возможности недоступны.

К счастью, решение подобных проблем несложно: достаточно перегрузить конструктор `Box()`, чтобы он учитывал только что описанные ситуации. Ниже приведена программа, которая содержит усовершенствованную версию класса `Box`, выполняющую эту задачу.

```
/* В этом примере класс Box определяет три конструктора для
   инициализации размеров параллелепипеда различными способами.
*/
```

```
class Box {
    double width;
    double height;
    double depth;

    // конструктор, используемый при указании всех измерений
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // конструктор, используемый, когда ни один из размеров не указан
    Box() {
        width = -1;  // значение -1 используется для указания
        height = -1; // неинициализированного
        depth = -1;  // параллелепипеда
    }

    // конструктор, используемый при создании куба
    Box(double len) {
        width = height = depth = len;
    }

    // вычисление и возврат объема
        double volume() {
            return width * height * depth;
        }
    }

    class OverloadCons {
        public static void main(String args[]) {
        // создание параллелепипедов с применением различных
        // конструкторов
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        double vol;

        // получение объема первого параллелепипеда
        vol = mybox1.volume();
        System.out.println("Объем mybox1 равен " + vol);

        // получение объема второго параллелепипеда
        vol = mybox2.volume();
        System.out.println("Объем mybox2 равен " + vol);

        // получение объема куба
        vol = mycube.volume();
        System.out.println("Объем mycube равен " + vol);
    }
}
```

Эта программа создает следующий вывод.

```
Объем mybox1 равен 3000.0
Объем mybox2 равен -1.0
Объем mycube равен 343.0
```

Как видите, соответствующий перегруженный конструктор вызывается в зависимости от параметров, указанных при выполнении оператора `new`.

Использование объектов в качестве параметров

До сих пор в качестве параметров методов мы использовали только простые типы. Однако передача методам объектов и вполне допустима, и достаточно распространена. Например, рассмотрим следующую короткую программу.

```
// Методам можно передавать объекты.
class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // возврат значения true, если параметр o равен вызывающему объекту
    boolean equals(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.equals(ob2));
        System.out.println("ob1 == ob3: " + ob1.equals(ob3));
    }
}
```

Эта программа создает следующий вывод.

```
ob1 == ob2: true
ob1 == ob3: false
```

Как видите, метод `equals()` внутри метода `Test` проверяет равенство двух объектов и возвращает результат этой проверки. То есть он сравнивает вызывающий объект с тем, который был ему передан. Если они содержат одинаковые значения, метод возвращает значение `true`. В противном случае он возвращает значение `false`. Обратите внимание на то, что параметр `o` в методе `equals()` указывает `Test` в качестве типа. Хотя `Test` — тип класса, созданный программой, он используется совершенно так же, как встроенные типы Java.

Одно из наиболее часто встречающихся применений объектов-параметров — в конструкторах. Часто приходится создавать новый объект так, чтобы вначале он не отличался от какого-то существующего объекта. Для этого потребуется определить конструктор, который в качестве параметра принимает объект его класса. Например, следующая версия класса `Box` позволяет выполнять инициализацию одного объекта другим.

```
// В этой версии Box допускает инициализацию одного объекта другим.
class Box {
    double width;
    double height;
    double depth;
```

```
    // Обратите внимание на этот конструктор.
    // Он использует объект типа Box.
    Box(Box ob) { // передача объекта конструктору
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // конструктор, используемый при указании всех измерений
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // конструктор, используемый, если ни одно из изменений не указано
    Box() {
        width = -1;  // значение -1 используется для указания
        height = -1; // неинициализированного
        depth = -1;  // параллелепипеда
    }

    // конструктор, используемый при создании куба
    Box(double len) {
        width = height = depth = len;
    }

    // вычисление и возврат объема
    double volume() {
        return width * height * depth;
    }
}

class OverloadCons2 {
    public static void main(String args[]) {
        // создание параллелепипедов 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);

        // создание копии объекта mybox1
        double vol;

        // получение объема первого параллелепипеда
        vol = mybox1.volume();
        System.out.println("Объем mybox1 равен " + vol);

        // получение объема второго параллелепипеда
        vol = mybox2.volume();
        System.out.println("Объем mybox2 равен " + vol);

        // получение объема куба
        vol = mycube.volume();
        System.out.println("Объем куба равен " + vol);

        // получение объема клона
        vol = myclone.volume();
```

```
        System.out.println("Объем клона равен " + vol);
    }
}
```

Как вы убедитесь, приступив к созданию собственных классов, чтобы объекты можно было создавать удобным и эффективным образом, нужно располагать множеством форм конструкторов.

Более пристальный взгляд на передачу аргументов

В общем случае существует два способа, которыми компьютерный язык может передавать аргументы подпрограмме. Первый способ — *вызов по значению*. При использовании этого подхода *значение* аргумента копируется в формальный параметр подпрограммы. Следовательно, изменения, выполненные в параметре подпрограммы, не влияют на аргумент. Второй способ передачи аргумента — *вызов по ссылке*. При использовании этого подхода параметру передается ссылка на аргумент (а не его значение). Внутри подпрограммы эта ссылка используется для обращения к реальному аргументу, указанному в вызове. Это означает, что изменения, выполненные в параметре, будут влиять на аргумент, использованный в вызове подпрограммы. Как вы убедитесь, хотя в Java применяется передача по значению для всех аргументов, конкретный эффект будет зависеть от того, передается ли базовый тип или ссылочный.

Когда методу передается элементарный тип, он передается по значению. Таким образом, создается копия аргумента, и все происходящее с параметром, который принимает аргумент, не оказывает влияния вне метода. Рассмотрим следующую программу.

```
// Элементарные 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);
    }
}
```

Вывод этой программы имеет следующий вид.

```
a и b перед вызовом: 15 20
a и b после вызова: 15 20
```

Как видите, выполняемые внутри метода `meth()` операции не влияют на значения переменных `a` и `b`, использованных в вызове. Их значения не изменились.

При передаче объекта методу ситуация изменяется коренным образом, поскольку по существу объекты передаются при вызове по ссылке. Следует помнить, что при создании переменной типа класса создается лишь ссылка на объект. Таким образом, при передаче этой ссылки методу, принимающий ее параметр будет ссылаться на тот же объект, на который ссылается аргумент. По сути, это означает, что объекты действуют, как если бы они передавались методам по ссылке. Изменения объекта внутри метода *влияют* на объект, использованный в качестве аргумента. Рассмотрим такую программу.

```
// Объекты передаются по ссылке на них.

class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // передача объекта
    void meth(Test o) {
        o.a *= 2;
        o.b /= 2;
    }
}

class PassObjRe {
    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);
    }
}
```

Эта программа создает следующий вывод.

```
ob.a и ob.b перед вызовом: 15 20
ob.a и ob.b после вызова: 30 10
```

Как видите, в данном случае действия внутри метода `meth()` влияют на объект, использованный в качестве аргумента.

Помните! Когда ссылка на объект передается методу, сама ссылка передается с использованием вызова по значению. Однако поскольку передаваемое значение ссылается на объект, копия этого значения все равно будет ссылаться на тот же объект, что и соответствующий аргумент.

Возврат объектов

Метод может возвращать любой тип данных, в том числе созданные типы классов. Например, в следующей программе метод `icrByTen()` возвращает объект,

в котором значение переменной a на 10 больше значения этой переменной 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 значения: "
                           + ob2.a);
    }
}
```

Эта программа создает следующий вывод.

```
ob1.a: 2
ob2.a: 12
ob2.a после второго увеличения значения: 22
```

Как видите, при каждом вызове метода `incrByTen()` программа создает новый объект и возвращает ссылку на него вызывающей процедуре.

Приведенная программа иллюстрирует еще один важный момент: поскольку память для всех объектов резервируется динамически с помощью оператора `new`, программисту не нужно беспокоиться о том, чтобы объект не вышел за пределы области видимости, так как выполнение метода, в котором он был создан, прекращается. Объект будет существовать до тех пор, пока где-либо в программе будет существовать ссылка на него. При отсутствии какой-либо ссылки на него объект будет уничтожен во время следующего сбора "мусора".

Рекурсия

В языке Java поддерживается *рекурсия*. Рекурсия — это процесс определения чего-либо в терминах самого себя. Применительно к программированию на языке Java, рекурсия — это атрибут, который позволяет методу вызывать самого себя. Такой метод называют *рекурсивным*.

Классический пример рекурсии — вычисление факториала числа. Факториал числа N — это произведение всех целых чисел от 1 до N. Например, факториал 3 равен $1 \times 2 \times 3 \times$, или 6. Вот как можно вычислить факториал, используя рекурсивный метод.

```
// Простой пример рекурсии.
class Factorial {
    // это 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));
    }
}
```

Вывод этой программы таков.

```
Факториал 3 равен 6
Факториал 4 равен 24
Факториал 5 равен 120
```

Для тех, кто не знаком с рекурсивными методами, работа метода `fact()` может быть не совсем понятна. Вот как работает этот метод. При вызове метода `fact()` с аргументом, равным 1, функция возвращает 1. В противном случае она возвращает произведение `fact(n-1)*n`. Для вычисления этого выражения программа вызывает метод `fact()` с аргументом 2. Это приведет к третьему вызову метода с аргументом, равным 1. Затем этот вызов возвратит значение 1, которое будет умножено на 2 (значение `n` во втором вызове метода). Этот результат (равный 2) возвращается исходному вызову метода `fact()` и умножается на 3 (исходное значение `n`). В результате мы получаем ответ, равный 6. В метод `fact()` можно было бы вставить вызовы метода `println()`, которые будут отображать уровень каждого вызова и промежуточные результаты.

Когда метод вызывает самого себя, новым локальным переменным и параметрам выделяется место в стеке и код метода выполняется с этими новыми начальными значениями. При каждом возврате из рекурсивного вызова старые локальные переменные и параметры удаляются из стека, и выполнение продолжается с момента вызова внутри метода. Рекурсивные методы выполняют действия, подобные выдвиганию и складыванию телескопа.

Из-за дополнительной перегрузки ресурсов, связанной с дополнительными вызовами функций, рекурсивные версии многих подпрограмм могут выполняться несколько медленнее их итерационных аналогов. Большое количество обращений к методу может вызвать переполнение стека. Поскольку параметры и локальные переменные сохраняются в стеке, а каждый новый вызов создает новые копии этих значений, это может привести к переполнению стека. В этом случае система времени выполнения Java будет передавать исключение. Однако, вероятно, об этом можно не беспокоиться, если только рекурсивная подпрограмма не начинает себя вести странным образом.

Основное преимущество применения рекурсивных методов состоит в том, что их можно использовать для создания версий некоторых алгоритмов, которые проще и понятнее их аналогов с использованием итерации. Например, алгоритм бы-

строй сортировки достаточно трудно реализовать итерационным методом. А некоторые типы алгоритмов, связанных с искусственным интеллектом, легче всего реализовать именно с помощью рекурсивных решений.

При использовании рекурсивных методов нужно позаботиться о том, чтобы где-либо в программе присутствовал оператор `if`, осуществляющий возврат из рекурсивного метода без выполнения рекурсивного вызова. В противном случае, будучи вызванным, метод никогда не выполнит возврат. Эта ошибка очень часто встречается при работе с рекурсией. Поэтому во время разработки советуем как можно чаще использовать вызовы метода `println()`, чтобы можно было следить за происходящим и прервать выполнение в случае ошибки.

Рассмотрим еще один пример рекурсии. Рекурсивный метод `printArray()` выводит первые `i` элементов массива `values`.

```
// Еще один пример рекурсии.

class RecTest {
    int values[];
    RecTest(int i) {
        values = new int[i];
    }

    // рекурсивное отображение элементов 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);
    }
}
```

Эта программа создает 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 вы уже знаете, инкапсуляция связывает данные с манипулирующим ими кодом. Однако инкапсуляция предоставляет еще один важный атрибут: *управление доступом*. Инкапсуляция позволяет управлять тем, какие части программы могут

получать доступ к членам класса. Управление доступом позволяет предотвращать злоупотребления. Например, предоставляя доступ к данным только при помощи четко определенного набора методов, можно предотвратить злоупотребление этими данными. Таким образом, если класс реализован правильно, он создает "черный ящик", который можно использовать, но внутренний механизм которого защищен от повреждения. Однако представленные ранее классы не полностью соответствуют этой цели. Рассмотрим, например, класс `Stack`, представленный в конце главы 6. Хотя методы `push()` и `pop()` действительно предоставляют управляемый интерфейс стека, этот интерфейс не обязателен для использования. То есть другая часть программы может обойти эти методы и обратиться к стеку непосредственно. Понятно, что в "плохих руках" эта возможность может приводить к проблемам. В этом разделе мы представим механизм, с помощью которого можно строго управлять доступом к различным членам класса.

Способ доступа к члену класса определяется *модификатором доступа*, присутствующим в его объявлении. В языке Java определен обширный набор модификаторов доступа. Некоторые аспекты управления доступом связаны, главным образом, с наследованием и пакетами. (Пакет – это, по сути, группировка классов.) Эти составляющие механизма управления доступом Java будут рассмотрены в следующих разделах. А пока начнем с рассмотрения применения управления доступом к отдельному классу. Когда основы управления доступом станут понятны, освоение других аспектов не представит особой сложности.

Модификаторами доступа Java являются `public` (открытый), `private` (закрытый) и `protected` (защищенный). Java определяет также уровень доступа, предоставляемый по умолчанию. Модификатор `protected` (защищенный) применяется только при использовании наследования. Остальные модификаторы доступа описаны далее в этой главе.

Начнем с определения модификаторов `public` и `private`. Когда к члену класса применен модификатор доступа `public`, он становится доступным для любого другого кода. Когда член класса указан как `private`, он доступен только другим членам этого же класса. Теперь вам должно быть понятно, почему методу `main()` всегда предшествует модификатор `public`. Этот метод вызывается кодом, расположенным вне данной программы, т.е. системой времени выполнения Java. При отсутствии модификатора доступа по умолчанию член класса считается открытым внутри своего собственного пакета, но недоступным для кода, расположенного вне этого пакета. (Пакеты рассматриваются в следующей главе.)

В уже разработанных нами классах все члены класса использовали режим доступа, определенный по умолчанию, который, по сути, является открытым. Однако, как правило, это не будет соответствовать реальным требованиям. Обычно доступ к данным класса необходимо ограничить, предлагая доступ только через методы. Кроме того, в ряде случаев придется определять закрытые методы класса.

Модификатор доступа предшествует остальной спецификации типа члена. То есть оператор объявления члена должен начинаться с модификатора доступа.

```
public int i;
private double j;

private int myMethod(int a, char b) { // ...
```

Чтобы влияние использования открытого и закрытого доступа было понятно, рассмотрим следующую программу.

```
/* Эта программа демонстрирует различие между модификаторами
   public и private.
*/
```

```
class Test {
    int a;              // доступ, определенный по умолчанию
    public int b;       // открытый доступ
    private int c;      // закрытый доступ

    // методы доступа к c
    void setc(int i) { // установка значения переменной c
        c = i;
    }
    int getc() {       // получение значения переменной c
        return c;
    }
}

class AccessTest {
    public static void main(String args[]) {
        Test ob = new Test();

        // Эти операторы правильны, a и b доступны непосредственно
        ob.a = 10;
        ob.b = 20;

        // Этот оператор неверен и может вызвать ошибку
     // ob.c = 100;    // Ошибка!

        // Доступ к объекту c должен осуществляться с
        // использованием методов его класса
        ob.setc(100); // OK
        System.out.println("a, b, и c: " + ob.a + " " + ob.b +
                           " " + ob.getc());
    }
}
```

Как видите, внутри класса `Test` использован метод доступа, заданный по умолчанию, что в данном примере равносильно указанию доступа `public`. Объект `b` явно указан как `public`. Объект `c` указан как закрытый. Это означает, что он недоступен для кода вне его класса. Поэтому внутри класса `AccessTest` объект `c` не может применяться непосредственно. Доступ к нему должен осуществляться с использованием его открытых методов `setc()` и `getc()`. Удаление символа комментария из начала строки

```
// ob.c = 100; // Ошибка!
```

сделало бы компиляцию этой программы невозможной из-за нарушений правил доступа.

В качестве более реального примера применения управления доступом рассмотрим следующую усовершенствованную версию класса `Stack`, код которого был приведен в конце главы 6.

```
// Этот класс определяет целочисленный стек, который может
// содержать 10 значений.
class Stack {
    /* Теперь переменные stck и tos являются закрытыми. Это означает,
       что они не могут быть случайно или намеренно
       изменены так, чтобы повредить стек.
    */
    private int stck[] = new int[10];
    private int tos;

    // Инициализация верхушки стека
```

```
    Stack() {
        tos = -1;
    }

    // Проталкивание элемента в стек
    void push(int item) {
        if(tos==9)
            System.out.println("Стек полон.");
        else
            stck[++tos] = item;
    }

    // Выталкивание элемента из стека
    int pop() {
        if(tos < 0) {
            System.out.println("Стек не загружен.");
            return 0;
        }
        else
            return stck[tos--];
    }
}
```

Как видите, теперь обе переменные: `stck`, содержащая стек, и `tos`, содержащая индекс верхушки стека, указаны как `private`. Это означает, что обращение к ним или их изменение могут осуществляться только через методы `push()` и `pop()`. Например, указание переменной `tos` как закрытой препятствует случайной установке другими частями программы ее значения выходящим за пределы конца массива `stck`.

Следующая программа — усовершенствованная версия класса `Stack`. Чтобы убедиться в том, что члены класса `stck` и `tos` действительно недоступны, попытайтесь удалить символы комментария из строк операторов.

```
class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        // проталкивание чисел в стек
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        // выталкивание этих чисел из стека
        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());

        // эти операторы недопустимы
     // mystack1.tos = -2;
     // mystack2.stck[3] = 100;
    }
}
```

Обычно методы будут обеспечивать доступ к данным, которые определены классом, но это не обязательно. Переменная экземпляра вполне может быть от-

крытой, если на то имеются веские причины. Например, для простоты переменные экземпляров в большинстве простых классов, созданных в этой книге, определены как открытые. Однако в большинстве классов, применяемых в реальных программах, манипулирование данными должно будет выполняться только с использованием методов. В следующей главе вернемся к теме управления доступом. Вы убедитесь, что управление доступом особенно важно при использовании наследования.

Что такое `static`

В некоторых случаях желательно определить член класса, который будет использоваться независимо от любого объекта этого класса. Обычно обращение к члену класса должно осуществляться только в сочетании с объектом его класса. Однако можно создать член класса, который может использоваться самостоятельно, без ссылки на конкретный экземпляр. Чтобы создать такой член, в начало его объявления нужно поместить ключевое слово `static`. Когда член класса объявлен как `static` (статический), он доступен до создания каких-либо объектов его класса и без ссылки на какой-либо объект. Статическими могут быть объявлены как методы, так и переменные. Наиболее распространенный пример статического члена — метод `main()`. Этот метод объявляют как `static`, поскольку он должен быть объявлен до создания любых объектов.

Переменные экземпляров, объявленные как `static`, по существу являются глобальными переменными. При объявлении объектов их класса программа не создает никаких копий статической переменной. Вместо этого все экземпляры класса совместно используют одну и ту же статическую переменную.

На методы, объявленные как `static`, накладывается ряд ограничений.

- Они могут непосредственно вызывать только другие статические методы.
- Они могут непосредственно осуществлять доступ только к статическим переменным.
- Они никоим образом не могут ссылаться на члены типа `this` или `super`. (Ключевое слово `super` связано с наследованием и описывается в следующей главе.)

Если для инициализации статических переменных нужно выполнить вычисления, можно объявить статический блок, который будет выполняться только один раз при первой загрузке класса. В следующем примере показан класс, который содержит статический метод, несколько статических переменных и статический блок инициализации.

```
// Демонстрация 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("Статический блок инициализирован.");
```

```
        b = a * 4;
    }

    public static void main(String args[]) {
        meth(42);
    }
}
```

Сразу после загрузки класса `UseStatic` программа выполняет все статические операторы. Вначале значение переменной `a` устанавливается равным `3`, затем программа выполняет статический блок, который выводит сообщение, и инициализирует переменную `b` значением `a*4`, или `12`. После этого программа вызывает метод `main()`, который обращается к методу `meth()`, передавая параметру `x` значение `42`. Три вызова метода `println()` ссылаются на две статические переменные `a` и `b`, а также на локальную переменную `x`.

Вывод этой программы таков.

```
Статический блок инициализирован.
x = 42
a = 3
b = 12
```

За пределами класса, в котором они определены, статические методы и переменные могут использоваться независимо от какого-либо объекта. Для этого достаточно указать имя их класса, за которым должен следовать точечный оператор. Например, если статический метод нужно вызвать извне его класса, это можно сделать используя следующую общую форму.

```
имя_класса.метод()
```

Здесь *имя_класса* — имя класса, в котором объявлен статический метод. Как видите, этот формат аналогичен применяемому для вызова нестатических методов через переменные объектных ссылок. Статическая переменная доступна аналогичным образом — с использованием точечного оператора, следующей за именем класса. Так в языке Java реализованы управляемые версии глобальных методов и переменных.

Приведем пример. Внутри метода `main()` обращение к статическому методу `callme()` и статической переменной `b` осуществляется с использованием имени 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 этой программы выглядит следующим образом.

```
a = 42
b = 99
```

Знакомство с ключевым словом final

Поле может быть объявлено как final (финальное). Это позволяет предотвратить изменение содержимого переменной, сделав ее, по сути, константой. Это означает, что финальное поле должно быть инициализировано во время его объявления. Значение можно также присвоить в пределах конструктора, но первый подход более распространен.

```
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 все последующие части программы могут пользоваться переменной FILE_OPEN и прочими так, как если бы они были константами, без риска изменения их значений. В практике программирования на Java принято идентификаторы всех финальных полей записывать прописными буквами, как в приведенном выше примере.

Кроме полей, как final могут быть объявлены параметры метода и локальные переменные. Объявление параметра как final препятствует его изменению в пределах метода. Объявление как final локальной переменной препятствует присвоению ей значения более одного раза.

Ключевое слово final можно применять также к методам, но в этом случае его значение существенно отличается от применяемого к переменным. Это дополнительное применение ключевого слова final описано в следующей главе, посвященной наследованию.

Повторное рассмотрение массивов

Массивы были представлены ранее в этой книге до того, как мы рассмотрели классы. Теперь, имея представление о классах, можно сделать важный вывод относительно массивов: все они реализованы как объекты. В связи с этим существует специальный атрибут массива, который наверняка пригодится. В частности, размер массива, т.е. количество элементов, которые может содержать массив, хранится в его переменной экземпляра length. Все массивы обладают этой переменной, которая всегда будет содержать размер массива. Ниже приведен пример программы, которая демонстрирует это свойство.

```
// Эта программа демонстрирует член длины 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);
    }
}
```

Эта программа создает следующий вывод.

```
длина a1 равна 10
длина a2 равна 8
длина a3 равна 4
```

Как видите, программа отображает размер каждого массива. Имейте в виду, что значение переменной `length` никак не связано с количеством действительно используемых элементов. Оно отражает лишь то количество элементов, которое может содержать массив.

Член `length` может находить применение во множестве ситуаций. Например, ниже показана усовершенствованная версия класса `Stack`. Как вы, возможно, помните, предшествующие версии этого класса всегда создавали 10-элементный стек. Следующая версия позволяет создавать стеки любого размера. Значение `stck.length` служит для предотвращения переполнения стека.

```
// Усовершенствованный класс Stack, в котором использован
// член длины массива.
class Stack {
    private int stck[];
    private int tos;

    // резервирование и инициализация стека
    Stack(int size) {
        stck = new int[size];
    tos = -1;
    }

    // Проталкивание элемента в стек
    void push(int item) {
        if(tos==stck.length-1) // использование члена длины массива
            System.out.println("Стек полон.");
        else
            stck[++tos] = item;
    }

    // Выталкивание элемента из стека
    int pop() {
        if(tos < 0) {
            System.out.println("Стек не загружен.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class TestStack2 {
    public static void main(String args[]) {
        Stack mystack1 = new Stack(5);
        Stack mystack2 = new Stack(8);

        // проталкивание чисел в стек
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);

        // выталкивание этих чисел из стека
        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 на то, что программа создает два стека: один глубиной в пять элементов, а второй — в шесть. Как видите, то, что массивы поддерживают информацию о своей длине, упрощает создание стеков любого размера.

Представление вложенных и внутренних классов

Язык Java позволяет определять класс внутри другого класса. Такие классы называют *вложенными классами*. Область видимости вложенного класса ограничена областью видимости внешнего класса. Таким образом, если класс B определен внутри класса A, класс B не может существовать независимо от класса A. Вложенный класс имеет доступ к членам (в том числе закрытым) класса, в который он вложен. Однако внешний класс не имеет доступа к членам вложенного класса. Вложенный класс, который объявлен непосредственно внутри области видимости своего внешнего класса, является его членом. Можно также объявлять вложенные классы, являющиеся локальными для блока.

Существует два типа вложенных классов: *статические* и *нестатические*. Статический вложенный класс — класс, к которому применен модификатор `static`. Поскольку он является статическим, должен обращаться к нестатическим членам своего внешнего класса при помощи объекта. То есть он не может непосредственно ссылаться на нестатические члены своего внешнего класса. Из-за этого ограничения статические вложенные классы используются редко.

Наиболее важный тип вложенного класса — *внутренний* класс. Внутренний класс — это нестатический вложенный класс. Он имеет доступ ко всем переменным и методам своего внешнего класса и может непосредственно ссылаться на них так же, как это делают остальные нестатические члены внешнего класса.

Следующая программа иллюстрирует определение и использование внутреннего класса. Класс `Outer` содержит одну переменную экземпляра `outer_x`, один метод экземпляра `test()` и определяет один внутренний класс `Inner`.

```
// Демонстрация использования внутреннего класса.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // это внутренний класс
    class Inner {
        void display() {
            System.out.println("вывод: outer_x = " + outer_x);
        }
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Это приложение создает следующий вывод.

```
вывод: outer_x = 100
```

В этой программе внутренний класс `Inner` определен в области видимости класса `Outer`. Поэтому любой код в классе `Inner` может непосредственно обращаться к переменной `outer_x`. Метод экземпляра `display()` определен внутри класса `Inner`. Этот метод отображает значение переменной `outer_x` в стандартном выходном потоке. Метод `main()` экземпляра `InnerClassDemo` создает экземпляр класса `Outer` и вызывает его метод `test()`. Этот метод создает экземпляр класса `Inner` и вызывает метод `display()`.

Важно понимать, что экземпляр класса `Inner` может быть создан только внутри области видимости класса `Outer`. Компилятор Java создает сообщение об ошибке, если любой код вне класса `Outer` пытается инициализировать класс `Inner`. В общем случае экземпляр внутреннего класса должен создаваться содержащей его областью.

Как уже было сказано, внутренний класс имеет доступ ко всем элементам своего внешнего класса, но не наоборот. Члены внутреннего класса известны только внутри области видимости внутреннего класса и не могут быть использованы внешним классом.

```
// Компиляция этой программы будет невозможна.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // это внутренний класс
    class Inner {
        int y = 10; // y - локальная переменная класса Inner

        void display() {
            System.out.println("вывод: outer_x = " + outer_x);
        }
    }

    void showy() {
        System.out.println(y); // ошибка, здесь переменная
                               // y не известна!
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

В этом примере переменная `y` объявлена как переменная экземпляра класса `Inner`. Поэтому она не известна за пределами класса и не может использоваться методом `showy()`.

Хотя мы уделили основное внимание внутренним классам, определенным в качестве членов внутри области видимости внешнего класса, внутренние классы можно определять внутри области видимости любого блока. Например, вложенный класс можно определить внутри блока, определенного методом, или даже внутри тела цикла `for`, как показано в следующем примере.

```
// Определение внутреннего класса внутри цикла for.
class Outer {
    int outer_x = 100;

    void test() {
        for(int i=0; i<10; i++) {
            class Inner {
                void display() {
                    System.out.println("вывод: outer_x = " + outer_x);
                }
            }
            Inner inner = new Inner();
            inner.display();
        }
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Вывод, создаваемый этой версией программы, показан ниже.

```
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
```

Хотя вложенные классы применимы не во всех ситуациях, они особенно удобны при обработке событий. Мы вернемся к теме вложенных классов в главе 22. В ней представлены внутренние классы, которые можно использовать для упрощения кода, предназначенного для обработки определенных типов событий. Читатели ознакомятся также с *анонимными внутренними классами*, являющимися внутренними классами без имен.

И последнее: первоначальная спецификация Java версии 1.0 не допускала использования вложенных классов. Они появились в версии Java 1.1.

Описание класса `String`

Хотя класс `String` подробно будет рассмотрен в части II этой книги, здесь уместно кратко ознакомить с ним читателей, поскольку мы будем использовать строки в некоторых последующих примерах части I. Вероятно, `String` — наиболее часто используемый класс из библиотеки классов Java. Очевидная причина этого в том, что строки — исключительно важный элемент программирования.

Во-первых, следует уяснить, что любая создаваемая строка в действительности представляет собой объект класса `String`. Даже строковые константы в действительности являются объектами класса `String`. Например, в операторе

```
System.out.println("Это - также объект String");
```

строка "Это - также объект String" — объект класса String.

Во-вторых, объекты класса String являются неизменяемыми. После того как он создан, его содержимое не может изменяться. Хотя это может показаться серьезным ограничением, на самом деле это не так по двум причинам.

- Если нужно изменить строку, всегда можно создать новую строку, содержащую все изменения.
- В Java определен класс StringBuffer, равноправный классу String, допускающий изменение строк, что позволяет выполнять в Java все обычные манипуляции строками. (Класс StringBuffer описан в части II.)

Существует множество способов создания строк. Простейший из них — воспользоваться оператором вроде следующего.

```
String myString = "тестовая строка";
```

Как только объект класса String создан, его можно использовать во всех ситуациях, в которых допустимо использование строк. Например, следующий оператор отображает содержимое объекта myString.

```
System.out.println(myString);
```

Для объектов класса String в Java определен один оператор, +, который служит для объединения двух строк. Например, оператор

```
String myString = "Мне" + " нравится " + "Java.";
```

приводит к тому, что содержимым переменной myString становится строка "Мне нравится Java".

Следующая программа иллюстрирует описанные концепции.

```
// Демонстрация применения строк.
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);
    }
}
```

Эта программа создает следующий вывод.

```
Первая строка
Вторая строка
Первая строка и вторая строка
```

Класс String содержит несколько методов, которые можно использовать. Опишем некоторые из них. С помощью метода equals() можно проверять равенство двух строк. Метод length() позволяет выяснить длину строки. Вызывая метод charAt(), можно получить символ с указанным индексом. Ниже приведены общие формы этих трех методов.

```
boolean equals(втораяСтр)
int length()
char charAt(индекс)
```

Следующая программа демонстрирует применение этих методов.

```
// Демонстрация некоторых 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 strOb1: " +
                           strOb1.length());

        System.out.println("Символ с индексом 3 в strOb1: " +
                           strOb1.charAt(3));

        if(strOb1.equals(strOb2))
            System.out.println("strOb1 == strOb2");
        else
            System.out.println("strOb1 != strOb2");

        if(strOb1.equals(strOb3))
            System.out.println("strOb1 == strOb3");
        else
            System.out.println("strOb1 != strOb3");
    }
}
```

Эта программа создает следующий вывод.

```
Длина strOb1: 12
Символ с индексом 3 в strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

Конечно, подобно тому, как могут существовать массивы любого другого типа объектов, могут существовать и массивы строк.

```
// Демонстрация использования массивов объектов типа 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]);
    }
}
```

Вывод этой программы таков.

```
str[0]: один
str[1]: два
str[2]: три
```

Как вы убедитесь из следующего раздела, строковые массивы играют важную роль во многих программах Java.

Использование аргументов командной строки

Иногда необходимо передать определенную информацию программе во время ее запуска. Для этого используют *аргументы командной строки* метода `main()`. Аргумент командной строки — это информация, которую во время запуска программы задают в командной строке непосредственно после ее имени. Доступ к ар-

гументам командной строки внутри программы Java не представляет сложности — они хранятся в виде строк в массиве типа `String`, переданного методу `main()`. Первый аргумент командной строки хранится в элементе массива `args[0]`, второй — в элементе `args[1]` и т.д. Например, следующая 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]);
    }
}
```

Попытайтесь выполнить эту программу, введя следующую строку.

```
java CommandLine this is a test 100 -1
```

В результате отобразится следующий вывод.

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

Помните! Все аргументы командной строки передаются как строки. Численные значения нужно вручную преобразовать в их внутренние представления, как поясняется в главе 16.

Список аргументов переменной длины

В JDK 5 была добавлена новая функциональная возможность, которая упрощает создание методов, принимающих переменное количество аргументов. Это средство получило название *vararg* (сокращение от *variable-length arguments* — список аргументов переменной длины). Метод, который принимает переменное количество аргументов, называют *методом с переменным количеством аргументов*.

Ситуации, в которых методу нужно передавать переменное количество аргументов, встречаются не так уж редко. Например, метод, который открывает подключение к Интернету, может принимать имя пользователя, пароль, имя файла, протокол и тому подобное, но применять значения, заданные по умолчанию, если какие-либо из этих сведений опущены. В этой ситуации было бы удобно передавать только те аргументы, для которых заданные по умолчанию значения не применимы. Еще один пример — метод `printf()`, входящий в состав библиотеки ввода-вывода Java. Как будет показано в главе 19, он принимает переменное количество аргументов, которые форматирует, а затем выводит.

До версии J2SE 5 обработка списка аргументов переменной длины могла выполняться двумя способами, ни один из которых не был особенно удобен. Во-первых, если максимальное количество аргументов было небольшим и известным, можно было создавать перегруженные версии метода — по одной для каждого возможного способа вызова метода. Хотя этот способ и приемлем, но применим только в редких случаях.

Во-вторых, когда максимальное количество возможных аргументов было большим или неизвестным, применялся подход, при котором аргументы сначала помещались в массив, а затем массив передавался методу. Следующая программа иллюстрирует этот подход.

```
// Использование массива для передачи методу переменного
// количества аргументов. Это старый стиль подхода
// к обработке списка аргументов 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 способ создания массива
        // 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
    }
}
```

Эта программа создает следующий вывод.

```
Количество аргументов: 1 Содержимое: 10
Количество аргументов: 3 Содержимое: 1 2 3
Количество аргументов: 0 Содержимое:
```

В программе методу `vaTest()` аргументы передаются через массив `v`. Этот старый подход к обработке списка аргументов переменной длины позволяет методу `vaTest()` принимать любое количество аргументов. Однако он требует, чтобы эти аргументы были вручную помещены в массив до вызова метода `vaTest()`. Создание массива при каждом вызове метода `vaTest()` — задача не только трудоемкая, но и чревата ошибками. Возможность использования методов с переменным количеством аргументов обеспечивает более простой и эффективный подход.

Для указания списка аргументов переменной длины используют три точки (...). Например, вот как метод `vaTest()` можно записать с использованием списка аргументов переменной длины.

```
static void vaTest(int ... v) {
```

Эта синтаксическая конструкция указывает компилятору, что метод `vaTest()` может вызываться без аргументов или с несколькими аргументами. В результате массив `v` неявно объявляется как массив типа `int[]`. Таким образом, внутри метода `vaTest()` доступ к массиву `v` осуществляется с использованием синтаксиса обычного массива. Предыдущая программа с применением метода с переменным количеством аргументов приобретает следующий вид.

```
// Демонстрация использования списка аргументов переменной длины.
class VarArgs {
    // теперь vaTest() использует список аргументов переменной длины.
    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 возможные способы вызова
        // vaTest() с переменным количеством аргументов.
        vaTest(10);      // 1 аргумент
        vaTest(1, 2, 3); // 3 аргумента
        vaTest();        // без аргументов
    }
}
```

Вывод этой программы совпадает с выводом исходной версии.

Отметим две важные особенности этой программы. Во-первых, как уже было сказано, внутри метода `vaTest()` переменная `v` действует как массив. Это обусловлено тем, что переменная `v` *является* массивом. Синтаксическая конструкция `...` просто указывает компилятору, что метод будет использовать переменное количество аргументов и что эти аргументы будут храниться в массиве, на который ссылается переменная `v`. Во-вторых, в методе `main()` метод `vaTest()` вызывается с различным количеством аргументов, в том числе и вовсе без аргументов. Аргументы автоматически помещаются в массив и передаются переменной `v`. В случае отсутствия аргументов длина массива равна нулю.

Наряду с параметром с переменным количеством аргументов массив может содержать "нормальные" параметры. Однако параметр с переменным количеством аргументов должен быть последним параметром, объявленным методом. Например, следующее объявление метода вполне допустимо.

```
int doIt(int a, int b, double c, int ... vals) {
```

В данном случае первые три аргумента, указанные в обращении к методу `doIt()`, соответствуют первым трем параметрам. Все остальные аргументы считаются принадлежащими параметру `vals`.

Помните, что параметр с переменным количеством аргументов должен быть последним. Например, следующее объявление записано неправильно:

```
int doIt(int a, int b, double c,
         int ... vals, boolean stopFlag) { // Ошибка!
```

В этом примере предпринимается попытка объявления обычного параметра после параметра с переменным количеством аргументов, что недопустимо. Существует еще одно ограничение, о котором следует знать: метод должен содержать только одни параметр с переменным количеством аргументов. Например, следующее объявление также неверно.

```
int doIt(int a, int b, double c,
         int ... vals, double ... morevals) { // Ошибка!
```

Попытка объявления второго параметра с переменным количеством аргументов недопустима. Рассмотрим измененную версию метода `vaTest()`, которая принимает обычный аргумент и список аргументов переменной длины.

```
// Использование списка аргументов переменной длины совместно
// со стандартными аргументами.
class VarArgs2 {

    // В этом примере msg - обычный параметр,
    // а v - параметр vararg.
```

```
    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 параметр vararg: ", 10);
        vaTest("Три параметра vararg: ", 1, 2, 3);
        vaTest("Без параметров vararg: ");
    }
}
```

Вывод этой программы таков.

```
Один параметр vararg: 1 Содержимое: 10
Три параметра vararg: 3 Содержимое: 1 2 3
Без параметров vararg: 0 Содержимое:
```

Перегрузка методов с переменным количеством аргументов

Метод, который принимает список аргументов переменной длины, можно перегружать.

```
// Параметры vararg 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("Проверка: ", 10, 20);
        vaTest(true, false, false);
    }
}
```

Эта программа создает следующий вывод.

```
vaTest(int ...): Количество аргументов: 3 Содержимое: 1 2 3
vaTest(String, int ...): Проверка: 2 Содержимое: 10 20
vaTest(boolean ...) Количество аргументов: 3 Содержимое: true
false false
```

Приведенная программа иллюстрирует два возможных способа перегрузки метода с переменным количеством аргументов. Первый способ — типы его параметра с переменным количеством аргументов могут быть различными. Именно это имеет место в вариантах `vaRest(int...)` и `vaTest(boolean...)`. Помните, что конструкция `...` вынуждает компилятор обрабатывать параметр как массив указанного типа. Поэтому, подобно тому как можно выполнять перегрузку методов, используя различные типы параметров массива, можно выполнять перегрузку методов с переменным количеством аргументов, используя различные типы списков аргументов переменной длины. В этом случае система Java использует различие в типах для определения нужного варианта перегруженного метода.

Второй способ перегрузки метода с переменным количеством аргументов — добавление одного или нескольких обычных параметров. Именно это было сделано для метода `vaTest(String, int...)`. В данном случае для определения нужного метода система Java использует и количество аргументов, и их тип.

На заметку! Метод, поддерживающий переменное количество аргументов, может быть перегружен также методом, который не поддерживает эту возможность. Например, в приведенной ранее программе метод `vaTest()` может быть перегружен методом `vaTest(int x)`. Эта специализированная версия вызывается только при наличии аргумента `int`. В случае передачи методу двух и более аргументов типа `int` программа будет использовать версию метода `vaTest(int...v)` с переменным количеством аргументов.

Переменное количество аргументов и неопределенность

При перегрузке метода, принимающего список аргументов переменной длины, могут случаться непредвиденные ошибки. Они связаны с неопределенностью, которая может возникать при вызове перегруженного метода со списком аргументов переменной длины. Например, рассмотрим следующую программу.

```
// Список аргументов переменной длины, перегрузка и неопределенность.
//
// Эта программа содержит ошибку, и ее компиляция
// будет невозможна!
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);            // OK
        vaTest(true, false, false); // OK
        vaTest();                   // Ошибка: неопределенность!
    }
}
```

В этой программе перегрузка метода `vaTest()` выполняется вполне корректно. Однако ее компиляция будет невозможна из-за следующего вызова.

```
vaTest(); // Ошибка: неопределенность!
```

Поскольку параметр с переменным количеством аргументов может быть пустым, этот вызов может быть преобразован в обращение к методу `vaTest(int...)` или `vaTest(boolean...)`. Оба варианта допустимы. Поэтому вызов принципиально неоднозначен.

Рассмотрим еще один пример неопределенности. Следующие перегруженные версии метода `vaTest()` изначально неоднозначны, несмотря на то что одна из них принимает обычный параметр.

```
static void vaTest(int ... v) { // ...
static void vaTest(int n, int ... v) { // ...
```

Хотя списки параметров метода `vaTest()` различны, компилятор не имеет возможности разрешения следующего вызова.

```
vaTest(1)
```

Должен ли он быть преобразован в обращение к методу `vaTest(int...)` с переменным количеством аргументов или в обращение к методу `vaTest(int, int...)` без переменного количества аргументов? Компилятор не имеет возможности ответить на этот вопрос. Таким образом ситуация неоднозначна.

Из-за ошибок неопределенности, подобных описанным, в некоторых случаях придется пренебрегать перегрузкой и просто использовать два различных имени метода. Кроме того, в некоторых случаях ошибки неопределенности служат признаком концептуальных изъянов программы, которые можно устранить за счет более тщательного построения решения задачи.

ГЛАВА 8

Наследование

Одним из фундаментальных понятий объектно-ориентированного программирования является наследование, поскольку оно позволяет создавать иерархические классификации. Используя наследование, можно создать общий класс, который определяет характеристики, общие для набора связанных элементов. Затем этот класс может наследоваться другими, более специализированными, классами, каждый из которых будет добавлять свои уникальные характеристики. В терминологии Java наследуемый класс называют *суперклассом*. Наследующий класс носит название *подкласса*. Следовательно, подкласс — это специализированная версия суперкласса. Он наследует все члены, определенные суперклассом, и добавляет собственные, уникальные, элементы.

Основы наследования

Чтобы наследовать класс, достаточно просто вставить определение одного класса в другой с использованием ключевого слова `extends`. В качестве иллюстрации рассмотрим короткий пример. Следующая программа создает суперкласс `A` и подкласс `B`. Обратите внимание на использование ключевого слова `extends` для создания подкласса класса `A`.

```
// Простой пример наследования.

// Создание суперкласса.
class A {
    int i, j;

    void showij() {
        System.out.println("i и j: " + i + " " + j);
    }
}

// Создание подкласса за счет 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 может использоваться самостоятельно.
        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();
    }
}
```

Эта программа создает следующий вывод.

```
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. Именно поэтому объект subOb имеет доступ к переменным i и j и может вызывать метод showij(). Кроме того, внутри метода sum() возможна непосредственная ссылка на переменные i и j, как если бы они были частью класса B.

Несмотря на то что A — суперкласс класса B, он также является полностью независимым, самостоятельным, классом. То, что класс является суперклассом подкласса, не означает невозможность его самостоятельного использования. Более того, подкласс может быть суперклассом другого подкласса.

Общая форма объявления класса, который наследуется от суперкласса, следующая.

```
class имя_подкласса extends имя_суперкласса {
    // тело класса
}
```

Для каждого создаваемого подкласса можно указывать только один суперкласс. Язык Java не поддерживает наследование нескольких суперклассов в одном подклассе. Как было сказано, можно создать иерархию наследования, в которой подкласс становится суперклассом другого подкласса. Однако никакой класс не может быть собственным суперклассом.

Доступ к членам и наследование

Хотя подкласс включает в себя все члены своего суперкласса, он не может получать доступ к тем членам суперкласса, которые объявлены как `private`. Например, рассмотрим следующую простую иерархию классов.

```
/* В иерархии классов закрытые члены остаются закрытыми
   для своего класса.

   Эта программа содержит ошибку, и ее компиляция
   будет невозможна.
*/

// Создание суперкласса.
class A {
    int i;          // открытая по умолчанию
    private int j; // закрытая для A

    void setij(int x, int y) {
        i = x;
        j = y;
    }
}

// Переменная j класса A в этом классе недоступна.
class B extends A {
    int total;

    void sum() {
        total = i + j; // ОШИБКА, j в этом классе недоступна
    }
}

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);
    }
}
```

Компиляция этой программы будет невозможна, поскольку использование переменной `j` внутри метода `sum()` класса `B` приводит к нарушению правил доступа. Поскольку переменная `j` объявлена как `private`, она доступна только другим членам ее собственного класса. Подкласс не имеет к ней доступа.

Помните! Член класса, который объявлен как закрытый, останется закрытым для своего класса. Он недоступен любому коду за пределами его класса, в том числе подклассам.

Более реальный пример

Рассмотрим более реальный пример, который поможет проиллюстрировать возможности наследования. В нем мы расширим последнюю версию класса `Box`, разработанную в предыдущей главе, добавив в нее четвертый компонент, который

назовем weight (вес). Таким образом, новый класс будет содержать ширину, высоту, глубину и вес параллелепипеда.

```
// В этой программе наследование используется для расширения класса Box.
class Box {
    double width;
    double height;
    double depth;

    // конструирование клона объекта
    Box(Box ob) { // передача объекта конструктору
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // конструктор, используемый при указании всех измерений
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // конструктор, используемый, если изменений нет
    Box() {
        width = -1;  // значение -1 используется для указания
        height = -1; // неинициализированного
        depth = -1;  // параллелепипеда
    }

    // конструктор, используемый при создании куба
    Box(double len) {
        width = height = depth = len;
    }

    // вычисление и возврат объема
    double volume() {
        return width * height * depth;
    }
}
// Расширение класса Box включением в него веса.
class BoxWeight extends Box {
    double weight; // вес параллелепипеда

    // конструктор 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);
    }
}
```

Эта программа создает следующий вывод.

```
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`. Классу `BoxWeight` не нужно воссоздавать все характеристики класса `Box`. Он может просто расширять класс `Box` в соответствии с конкретными целями.

Основное преимущество наследования состоит в том, что как только суперкласс, который определяет общие атрибуты набора объектов, создан, его можно использовать для создания любого количества более специализированных классов. Каждый подкласс может точно определять свою собственную классификацию. Например, следующий класс наследует характеристики класса `Box` и добавляет атрибут цвета.

```
// Этот код расширяет класс Box, включая в него атрибут цвета.
class ColorBox extends Box {
    int color; // цвет параллелепипеда

    ColorBox(double w, double h, double d, int c) {
        width = w;
        height = h;
        depth = d;
        color = c;
    }
}
```

Помните, как только суперкласс, который определяет общие аспекты объекта, создан, он может наследоваться для создания специализированных классов. Каждый подкласс добавляет собственные уникальные атрибуты. В этом заключается сущность наследования.

Переменная суперкласса может ссылаться на объект подкласса

Ссылочной переменной суперкласса может быть присвоена ссылка на любой подкласс, производный от данного суперкласса. Этот аспект наследования будет весьма полезен во множестве ситуаций. Рассмотрим следующий пример.

```
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();
```

```
        // присваивание ссылке на объект BoxWeight ссылки на объект Box
        plainbox = weightbox;
        vol = plainbox.volume(); // OK, метод volume() определен в Box
        System.out.println("Объем plainbox равен " + vol);

        /* Следующий оператор ошибочен, поскольку plainbox
           не определяет член weight. */
        // System.out.println("Вес plainbox равен " + plainbox.weight);
    }
}
```

В этом примере `weightbox` — ссылка на объекты класса `BoxWeight`, а `plainbox` — ссылка на объекты класса `Box`. Поскольку `BoxWeight` — подкласс класса `Box`, ссылке `plainbox` можно присваивать ссылку на объект `weightbox`.

Важно понимать, что доступные объекты определяются типом ссылочной переменной, а не типом объекта, на который она ссылается. То есть при присваивании ссылочной переменной суперкласса ссылки на объект подкласса доступ предоставляется только к указанным в ней частям объекта, определенного суперклассом. Именно поэтому объект `plainbox` не имеет доступа к переменной `weight` даже в том случае, когда он ссылается на объект класса `BoxWeight`. Если немного подумать, это становится понятным — суперклассу не известно, что именно подкласс добавляет в него. Поэтому последняя строка кода в предыдущем фрагменте оформлена в виде комментария. Ссылка объекта класса `Box` не имеет доступа к полю `weight`, поскольку оно не определено в классе `Box`.

Хотя все сказанное выше может казаться несколько запутанным, эти особенности находят ряд практических применений, два из которых рассматриваются в последующих разделах данной главы.

Использование ключевого слова `super`

В предшествующих примерах классы, производные от класса `Box`, были реализованы не столь эффективно и надежно, как могли бы. Например, конструктор `BoxWeight()` явно инициализирует поля `width`, `height` и `depth` класса `Box`. Это не только ведет к дублированию кода суперкласса, что весьма неэффективно, но и предполагает наличие у подкласса доступа к этим членам. Однако в ряде случаев придется создавать суперкласс, подробности реализации которого доступны только для него самого (т.е. с закрытыми членами данных). В этом случае подкласс никак не может самостоятельно непосредственно обращаться к этим переменным или инициализировать их. Поскольку инкапсуляция — один из главных атрибутов ООП, не удивительно, что язык Java предлагает решение этой проблемы. Во всех случаях, когда подклассу нужно сослаться на его непосредственный суперкласс, это можно сделать при помощи ключевого слова `super`.

Ключевое слово `super` имеет две общие формы. Первую используют для вызова конструктора суперкласса, а вторую — для обращения к члену суперкласса, скрытому членом подкласса. Рассмотрим обе формы.

Использование ключевого слова `super` для вызова конструкторов суперкласса

Подкласс может вызывать конструктор, определенный его суперклассом, с помощью следующей формы ключевого слова `super`.

```
super(список_аргументов);
```

Здесь *список_аргументов* определяет любые аргументы, требуемые конструктору в суперклассе. Вызов метода `super()` всегда должен быть первым оператором, выполняемым внутри конструктора подкласса.

В качестве иллюстрации использования метода `super()` рассмотрим следующую усовершенствованную версию класса `BoxWeight`.

```
// Теперь класс BoxWeight использует ключевое слово super
// для инициализации своих атрибутов объекта Box.
class BoxWeight extends Box {
    double weight; // вес параллелепипеда

    // инициализация переменных width, height и depth с помощью super()
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // вызов конструктора суперкласса
        weight = m;
    }
}
```

В этом примере метод `BoxWeight()` вызывает метод `super()` с аргументами `w`, `h` и `d`. Это приводит к вызову конструктора `Box()`, который инициализирует переменные `width`, `height` и `depth`, используя переданные ему значения соответствующих параметров. Теперь класс `BoxWeight` не инициализирует эти значения самостоятельно. Ему нужно инициализировать только свое уникальное значение — `weight`. В результате при необходимости эти значения могут оставаться закрытыми значениями класса `Box`.

В приведенном примере метод `super()` был вызван с тремя аргументами. Поскольку конструкторы могут быть перегруженными, метод `super()` можно вызывать, используя любую форму, определенную суперклассом. Программа выполнит тот конструктор, который соответствует указанным аргументам. В качестве примера приведем полную реализацию класса `BoxWeight`, которая предоставляет конструкторы для различных способов создания параллелепипедов. В каждом случае метод `super()` вызывается с соответствующими аргументами. Обратите внимание на то, что внутри класса `Box` его члены `width`, `height` и `depth` объявлены как закрытые.

```
// Полная реализация класса BoxWeight.
class Box {
    private double width;
    private double height;
    private double depth;

    // конструирование клона объекта
    Box(Box ob) { // передача объекта конструктору
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // конструктор, используемый при указании всех измерений
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // конструктор, используемый, если ни одно из измерений не указано
    Box() {
        width = -1;  // значение -1 используется для указания
```

```
        height = -1; // неинициализированного
        depth = -1;  // параллелепипеда
    }

    // конструктор, используемый при создании куба
    Box(double len) {
        width = height = depth = len;
    }

    // вычисление и возврат объема
    double volume() {
        return width * height * depth;
    }
}

// Теперь BoxWeight полностью реализует все конструкторы.
class BoxWeight extends Box {
    double weight; // вес параллелепипеда

    // конструирование клона объекта
    BoxWeight(BoxWeight ob) { // передача объекта конструктору
        super(ob);
        weight = ob.weight;
    }

    // конструктор, используемый при указании всех параметров
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // вызов конструктора суперкласса
        weight = m;
    }

    // конструктор, используемый по умолчанию
    BoxWeight() {
        super();
        weight = -1;
    }

    // конструктор, используемый при создании куба
    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(); // по умолчанию
        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 of 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();
    }
}
```

Эта программа создает следующий вывод.

```
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 классе `BoxWeight`.

```
// конструирование клона объекта
BoxWeight(BoxWeight ob) { // передача объекта конструктору
    super(ob);
    weight = ob.weight;
}
```

Обратите внимание на то, что метод `super()` выполняет передачу объекту класса `BoxWeight`, а не класса `Box`. Тем не менее это все равно ведет к вызову конструктора `Box(Box.ob)`. Как уже было отмечено, переменную суперкласса можно использовать для ссылки на любой объект, унаследованный от этого класса. Таким образом, объект класса `BoxWeight` можно передать конструктору `Box()`. Конечно, классу `Box` будут известны только его собственные члены.

Рассмотрим основные концепции применения метода `super()`. Когда подкласс вызывает метод `super()`, он вызывает конструктор своего непосредственного суперкласса. Таким образом, метод `super()` всегда ссылается на суперкласс, расположенный в иерархии непосредственно над вызывающим классом. Это справедливо даже в случае многоуровневой иерархии. Кроме того, метод `super()` всегда должен быть первым оператором, выполняемым внутри конструктора подкласса.

Второе применение ключевого слова `super`

Вторая форма ключевого слова `super` действует подобно ключевому слову `this`, за исключением того, что всегда ссылается на суперкласс подкласса, в котором она использована. Общая форма этого применения ключевого слова `super` имеет следующий вид.

```
super.член
```

Здесь *член* может быть методом либо переменной экземпляра.

Вторая форма применения ключевого слова super наиболее подходит в тех ситуациях, когда имена членов подкласса скрывают члены суперкласса с такими же именами. Рассмотрим следующую простую иерархию классов.

```
// Использование ключевого слова super для предотвращения сокрытия имени.
class A {
    int i;
}

// Создание подкласса за счет расширения класса A.
class B extends A {
    int i;          // эта переменная i скрывает переменную i в классе A

    B(int a, int b) {
        super.i = a; // i в классе A
        i = b;       // i в классе 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();
    }
}
```

Эта программа отображает следующее.

```
i в суперклассе: 1
i в подклассе: 2
```

Хотя переменная экземпляра i в классе B скрывает переменную i в классе A, ключевое слово super позволяет получить доступ к переменной i, определенной в суперклассе. Как вы увидите, ключевое слово super можно использовать также для вызова методов, которые скрываются подклассом.

Создание многоуровневой иерархии

До сих пор мы использовали простые иерархии классов, которые состояли только из суперкласса и подкласса. Однако можно строить иерархии, которые содержат любое количество уровней наследования. Как уже отмечалось, вполне допустимо использовать подкласс в качестве суперкласса другого подкласса. Например, класс C может быть подклассом класса B, который, в свою очередь, является подклассом класса A. В подобных ситуациях каждый подкласс наследует все характеристики всех его суперклассов. В приведенном примере класс C наследует все характеристики классов B и A. В качестве примера многоуровневой иерархии рассмотрим следующую программу. В ней подкласс BoxWeight использован в качестве суперкласса для создания подкласса Shipment. Класс Shipment наследует все характеристики классов BoxWeight и Box, а также добавляет поле cost, которое содержит стоимость поставки такого пакета.

```
// Расширение класса BoxWeight за счет включения в него
// стоимости доставки.
```

```
// Начнем с создания класса Box.
class Box {
    private double width;
    private double height;
    private double depth;

    // конструирование клона объекта
    Box(Box ob) { // передача объекта конструктору
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // конструктор, используемый при указании всех измерений
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // конструктор, используемый, когда ни одно из измерений не указано
    Box() {
        width = -1;  // значение -1 используется для указания
        height = -1; // неинициализированного
        depth = -1;  // параллелепипеда
    }

    // конструктор, используемый при создании куба
    Box(double len) {
        width = height = depth = len;
    }

    // вычисление и возврат объема
    double volume() {
        return width * height * depth;
    }
}

// Добавление веса.
class BoxWeight extends Box {
    double weight; // вес параллелепипеда

    // конструирование клона объекта
    BoxWeight(BoxWeight ob) { // передача объекта конструктору
        super(ob);
        weight = ob.weight;
    }

    // конструктор, используемый при указании всех параметров
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // вызов конструктора суперкласса
        weight = m;
    }

    // конструктор, используемый по умолчанию
    BoxWeight() {
        super();
        weight = -1;
    }
```

```
    // конструктор, используемый при создании куба
    BoxWeight(double len, double m) {
        super(len);
        weight = m;
    }
}

// Добавление стоимости доставки.
class Shipment extends BoxWeight {
    double cost;

    // конструирование клона объекта
    Shipment(Shipment ob) { // передача объекта конструктору
        super(ob);
        cost = ob.cost;
    }

    // конструктор, используемый при указании всех параметров
    Shipment(double w, double h, double d,
    double m, double c) {
        super(w, h, d, m); // вызов конструктора суперкласса
        cost = c;
    }

    // конструктор, используемый по умолчанию
    Shipment() {
        super();
        cost = -1;
    }

    // конструктор, используемый при создании куба
    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);
    }
}
```

Вывод этой программы будет следующим.

```
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 использовать ранее определенные классы `Box` и `BoxWeight`, добавляя только ту дополнительную информацию, которая требуется для его собственного специализированного применения. В этом состоит одно из ценных свойств наследования. Оно позволяет повторно использовать код.

Приведенный пример иллюстрирует важный аспект: метод `super()` всегда ссылается на конструктор ближайшего суперкласса в иерархии. Метод `super()` в классе `Shipment` вызывает конструктор класса `BoxWeight`. Метод `super()` в классе `BoxWeight` вызывает конструктор класса `Box`. Если в иерархии классов конструктор суперкласса требует передачи ему параметров, все подклассы должны передавать эти параметры "по эстафете". Данное утверждение справедливо независимо от того, нуждается ли подкласс в собственных параметрах.

На заметку! В приведенном примере программы вся иерархия классов, включая `Box`, `BoxWeight` и `Shipment`, находится в одном файле. Это сделано только ради удобства. В Java все три класса могли бы быть помещены в отдельные файлы и компилироваться независимо друг от друга. Фактически использование отдельных файлов — норма, а не исключение при создании иерархий классов.

Порядок вызова конструкторов

В каком порядке вызываются конструкторы классов, образующих иерархию, при ее создании? Например, какой конструктор вызывается раньше: `A()` или `B()`, если `B` — это подкласс, а `A` — суперкласс? В иерархии классов конструкторы вызываются в порядке наследования, начиная с суперкласса и заканчивая подклассом. Более того, поскольку метод `super()` должен быть первым оператором, выполняемым в конструкторе подкласса, этот порядок остается неизменным, независимо от того, используется ли форма `super()`. Если метод `super()` не применяется, программа использует конструктор каждого суперкласса, заданный по умолчанию или не содержащий параметров. В следующей программе демонстрируется порядок выполнения конструкторов.

```
// Демонстрация порядка вызова конструкторов.
// Создание суперкласса.

class A {
    A() {
        System.out.println("Внутри конструктора A.");
    }
}

// Создание подкласса за счет расширения класса A.
class B extends A {
    B() {
        System.out.println("Внутри конструктора B.");
    }
}

// Создание еще одного подкласса за счет 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();
    }
}
```

Эта программа создает следующий вывод.

```
Внутри конструктора A
Внутри конструктора B
Внутри конструктора C
```

Как видите, конструкторы вызываются в порядке наследования.

Если немного подумать, становится ясно, что выполнение конструкторов в порядке наследования имеет смысл. Поскольку суперкласс ничего не знает о своих подклассах, любая инициализация, которую он должен выполнить, полностью независима и, возможно, обязательна для выполнения любой инициализацией, выполняемой подклассом. Поэтому она должна выполняться первой.

Переопределение методов

Если в иерархии классов имя и сигнатура типа метода подкласса совпадает с атрибутами метода суперкласса, говорят, что метод подкласса *переопределяет* метод суперкласса. Когда переопределенный метод вызывается из своего подкласса, он всегда будет ссылаться на версию этого метода, определенную подклассом. Версия метода, определенная суперклассом, будет скрыта. Рассмотрим следующий пример.

```
// Переопределение метода.
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }

    // отображение 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;
    }

    // отображение k - этот метод переопределяет метод show() класса 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(); // этот оператор вызывает метод show() класса B
    }
}
```

Эта программа создает следующий вывод.

```
k: 3
```

Когда программа вызывает метод `show()` по отношению к объекту типа `B`, она использует версию этого метода, определенную внутри класса `B`. То есть версия метода `show()`, определенная внутри класса `B`, переопределяет версию, объявленную внутри класса `A`.

Если нужно получить доступ к версии переопределенного метода, определенного в суперклассе, это можно сделать с помощью ключевого слова `super`. Например, в следующей версии класса `B` версия метода `show()`, объявленная в суперклассе, вызывается внутри версии подкласса. Это позволяет отобразить все переменные экземпляров.

```
class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    void show() {
        super.show(); // этот оператор вызывает метод show() класса A
        System.out.println("k: " + k);
    }
}
```

Подстановка этой версии класса `A` в предыдущую программу приведет к следующему выводу.

```
i и j: 1 2
k: 3
```

В этой версии метод `super.show()` вызывает версию метода `show()`, определенную в суперклассе.

Переопределение метода выполняется *только* в том случае, если имена и сигнатуры типов двух методов идентичны. В противном случае два метода являются просто перегруженными. Например, рассмотрим измененную версию предыдущего примера.

```
// Методы с различающимися сигнатурами являются
// перегруженными, а не переопределенными.
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    // отображение i и j
    void show() {
        System.out.println("i и j: " + i + " " + j);
    }
}
```

```
// Создание подкласса за счет расширения класса A.
class B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    // перегрузка метода 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: "); // вызов метода show() класса B
        subOb.show();          // вызов метода show() класса A
    }
}
```

Эта программа создает следующий вывод.

```
Это k: 3
i и j: 1 2
```

Версия метода `show()`, определенная в классе `B`, принимает строковый параметр. В результате ее сигнатура типа отличается от сигнатуры метода в классе `B`, который не принимает никаких параметров. Поэтому никакое переопределение (или сокрытие имени) не происходит. Вместо этого просто выполняется перегрузка версии метода `show()`, определенной в классе `A`, версией, определенной в классе `B`.

Динамическая диспетчеризация методов

Хотя приведенные в предыдущем разделе примеры демонстрируют механизм переопределения методов, они не показывают всех возможностей. Действительно, если бы переопределение методов служило лишь для удобства работы с пространством имен, оно представляло бы только определенный теоретический интерес и имело бы очень небольшое практическое значение. Однако это не так. Переопределение методов служит основой для одной из наиболее мощных концепций Java — *динамической диспетчеризации методов*. Динамическая диспетчеризация методов — механизм, за счет которого осуществляется поиск подходящей версии при обращении к переопределенному методу во время выполнения, а не компиляции.

Динамическая диспетчеризация методов важна потому, что именно с ее помощью Java реализует полиморфизм времени выполнения.

Рассмотрение этой концепции начнем с повторной формулировки одного важного принципа: ссылочная переменная суперкласса может ссылаться на объект подкласса. Система Java использует этот факт для разрешения обращений к переопределенным методам во время выполнения. Вот как это происходит. Когда вызов переопределенного метода реализуется с использованием ссылки на суперкласс, Java выбирает нужную версию этого метода в зависимости от типа объекта ссылки в момент вызова. Таким образом, этот выбор осуществляется во время выполнения. При ссылке на различные типы объектов программа будет обращаться

к различным версиям переопределенного метода. Иначе говоря, выбор для выполнения версии переопределенного метода осуществляется в зависимости от *типа объекта ссылки* (а не от типа ссылочной переменной). Следовательно, если суперкласс содержит метод, переопределяемый подклассом, то при наличии ссылки на различные типы объектов через ссылочную переменную суперкласса программа будет выполнять различные версии метода.

В следующем примере иллюстрируется динамическая диспетчеризация методов.

```
// Динамическая диспетчеризация методов
class A {
    void callme() {
        System.out.println("Внутри метода callme класса A");
    }
}

class B extends A {
    // переопределение метода callme()
    void callme() {
        System.out.println("Внутри метода callme класса B");
    }
}

class C extends A {
    // переопределение метода callme()
    void callme() {
        System.out.println("Внутри метода callme класса C");
    }
}

class Dispatch {
    public static void main(String args[]) {
        A a = new A(); // объект класса A
        B b = new B(); // объект класса B
        C c = new C(); // объект класса C

        A r;           // получение ссылки класса A

        r = a;         // r ссылается на объект A
        r.callme();    // вызов версии метода callme, определенной в A

        r = b;         // r ссылается на объект B
        r.callme();    // вызов версии метода callme, определенной в B

        r = c;         // r ссылается на объект C
        r.callme();    // вызов версии метода callme, определенной в C
    }
}
```

Эта программа создает следующий вывод.

```
Внутри метода callme класса A
Внутри метода callme класса B
Внутри метода callme класса C
```

Эта программа создает один суперкласс `A` и два его подкласса `B` и `C`. Подклассы `B` и `C` переопределяют метод `callme()`, объявленный в классе `A`. Внутри метода `main()` программа объявляет объекты классов `A`, `B` и `C`. Программа объявляет также ссылку класса `A` по имени `r`. Затем программа по очереди присваивает переменной `r` ссылку на каждый класс объекта и использует эту ссылку для вызова метода `callme()`. Как видно из вывода, выполняемая версия метода `callme()` определяется по классу объекта ссылки во время выполнения. Если бы выбор осуществлял-

ся по типу ссылочной переменной, r, вывод отражал бы три обращения к методу callme() класса A.

На заметку! Те читатели, которые знакомы с языками C++ или C#, должны заметить, что переопределенные методы в Java подобны виртуальным функциям в этих языках.

Для чего нужны переопределенные методы

Как уже было сказано, переопределенные методы позволяют Java поддерживать полиморфизм времени выполнения. Большое значение полиморфизма для объектно-ориентированного программирования обусловлено следующей причиной: он позволяет общему классу указывать методы, которые станут общими для всех его производных классов, в то же время позволяя подклассам определять конкретные реализации некоторых или всех этих методов. Переопределенные методы — еще один используемый в Java способ реализации аспекта полиморфизма под названием "один интерфейс, множество методов".

Одно из основных условий успешного применения полиморфизма — понимание того, что суперклассы и подклассы образуют иерархию по степени увеличения специализации. В случае его правильного применения суперкласс предоставляет все элементы, которые подкласс может использовать непосредственно. Он определяет также те методы, которые производный класс должен реализовать самостоятельно. Это позволяет подклассу определять собственные методы при сохранении единообразия интерфейса. Таким образом, объединяя наследование и переопределенные методы, суперкласс может определять общую форму методов, которые будут использоваться всеми его подклассами.

Динамический, реализуемый во время выполнения полиморфизм — один из наиболее мощных механизмов объектно-ориентированной архитектуры, обеспечивающих повторное использование и надежность кода. Возможность существующих библиотек кода вызывать методы применительно к экземплярам новых классов без повторной компиляции при сохранении четкого абстрактного интерфейса — чрезвычайно мощное средство.

Использование переопределения методов

Рассмотрим более реальный пример использования переопределения методов. Следующая программа создает суперкласс Figure, который хранит размеры двухмерного объекта. Она определяет также метод area(), который вычисляет площадь объекта. Программа создает два класса, производных от класса Figure, — Rectangle и Triangle. Каждый из этих подклассов переопределяет метод area(), чтобы он возвращал соответственно площадь четырехугольника и треугольника.

```
// Применение полиморфизма времени выполнения.
class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    double area() {
        System.out.println("Площадь фигуры не определена.");
```

```
        return 0;
    }
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // переопределение метода area для четырехугольника
    double area() {
        System.out.println("В области четырехугольника.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // переопределение метода area для прямоугольного треугольника
    double area() {
        System.out.println("В области треугольника.");
        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());
    }
}
```

Эта программа создает следующий вывод.

```
В области четырехугольника.
Площадь равна 45
В области треугольника.
Площадь равна 40
Область фигуры не определена.
Площадь равна 0
```

Двойственный механизм наследования и полиморфизма времени выполнения позволяет определить единый интерфейс, используемый несколькими различными, но сходными классами объектов. В данном случае, если объект является производным от класса `Figure`, его площадь можно вычислять, вызывая метод `area()`. Интерфейс выполнения этой операции остается неизменным, независимо от типа фигуры.

Использование абстрактных классов

В ряде ситуаций нужно будет определять суперкласс, который объявляет структуру определенной абстракции без предоставления полной реализации каждого метода. То есть иногда придется создавать суперкласс, определяющий только обобщенную форму, которую будут совместно использовать все его подклассы, добавляя необходимые детали. Такой класс определяет сущность методов, которые должны реализовать подклассы. Например, такая ситуация может возникать, когда суперкласс не в состоянии создать полноценную реализацию метода. Именно такая ситуация имела место в классе `Figure` в предыдущем примере. Определение метода `area()` — просто шаблон. Он не будет вычислять и отображать площадь объекта какого-либо типа.

Как вы убедитесь в процессе создания собственных библиотек классов, отсутствие полного определения метода в контексте суперкласса — не столь уж редкая ситуация. Эту проблему можно решать двумя способами. Один из них, как было показано в предыдущем примере, — просто вывод предупреждающего сообщения. Хотя этот подход и полезен в определенных ситуациях — например, при отладке, — обычно он не годится. Могут существовать методы, которые должны быть переопределены подклассом, чтобы подкласс имел какой-либо смысл. Рассмотрим класс `Triangle`. Он лишен всякого смысла, если метод `area()` не определен. В этом случае необходим способ убедиться в том, что подкласс действительно переопределяет все необходимые методы. В Java для этого служит *абстрактный метод*.

Потребовать, чтобы определенные методы переопределялись подклассом, можно с использованием указания модификатора типа `abstract`. Иногда такие методы называют относящимися к *компетенции подкласса*, поскольку в суперклассе для них никакой реализации не предусмотрено. Таким образом, подкласс должен переопределять эти методы — он не может просто использовать версию, определенную в суперклассе. Для объявления абстрактного метода используют следующую общую форму.

```
abstract тип имя(список_параметров);
```

Как видите, в этой форме тело метода отсутствует.

Любой класс, который содержит один или более абстрактных методов, должен быть также объявлен как абстрактный. Для этого достаточно поместить ключевое слово `abstract` перед ключевым словом `class` в начале объявления класса. Абстрактный класс не может содержать какие-то объекты. То есть абстрактный класс не может быть непосредственно конкретизирован с помощью оператора `new`. Такие объекты были бы бесполезны, поскольку абстрактный класс определен не полностью. Нельзя также объявлять абстрактные конструкторы или абстрактные статические методы. Любой подкласс абстрактного класса должен либо реализовать все абстрактные методы суперкласса, либо сам быть объявлен абстрактным.

Ниже приведен простой пример класса, содержащего абстрактный метод, и класса, который реализует этот метод.

```
// Простой пример применения абстракции.
abstract class A {
    abstract void callme();

    // абстрактные классы все же могут содержать конкретные методы
    void callmetoo() {
        System.out.println("Это конкретный метод.");
    }
}
```

```
class B extends A {
    void callme() {
        System.out.println("Реализация метода callme класса B.");
    }
}

class AbstractDemo {
    public static void main(String args[]) {
        B b = new B();

        b.callme();
        b.callmetoo();
    }
}
```

Обратите внимание на то, что в этой программе класс `A` не содержит объявлений каких-либо объектов. Как уже было сказано, конкретизация абстрактного класса невозможна. И еще один нюанс: класс `A` реализует конкретный метод `callmetoo()`. Это вполне допустимо. Абстрактные классы могут содержать любое необходимое количество конкретных реализаций.

Хотя абстрактные классы не могут быть использованы для конкретизации объектов, их можно применять для создания ссылок на объекты, поскольку в Java полиморфизм времени выполнения реализован с использованием ссылок на суперкласс. Поэтому должна существовать возможность создания ссылки на абстрактный класс, которая может использоваться для указания на объект подкласса. Применение этого свойства показано в следующем примере.

Используя абстрактный класс, можно усовершенствовать созданный ранее класс `Figure`. Поскольку понятие площади неприменимо к неопределенной двухмерной фигуре, следующая версия программы объявляет метод `area()` внутри класса `Figure` как `abstract`. Конечно, это означает, что все классы, производные от класса `Figure`, должны переопределять метод `area()`.

```
// Использование абстрактных методов и классов.
abstract class Figure {
    double dim1;
    double dim2;
    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    // теперь метод area является абстрактным
    abstract double area();
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // переопределение метода area для четырехугольника
    double area() {
        System.out.println("В области четырехугольника.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
```

```
        super(a, b);
    }

    // переопределение метода area для четырехугольника
    double area() {
        System.out.println("В области треугольника.");
        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; // этот оператор допустим,
                       // никакой 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 метода `main()`, объявление объектов типа `Figure` более недопустимо, поскольку теперь этот класс является абстрактным. И все подклассы класса `Figure` должны переопределять метод `area()`. Чтобы убедиться в этом, попытайтесь создать подкласс, который не переопределяет метод `area()`. Это приведет к ошибке времени компиляции.

Хотя создание объекта типа `Figure` недопустимо, можно создать ссылочную переменную типа `Figure`. Переменная `figref` объявлена как ссылка на тип `Figure`, т.е. ее можно использовать для ссылки на объект любого класса, производного от класса `Figure`. Как мы уже поясняли, поиск версий переопределенных методов во время выполнения осуществляется за счет ссылки на суперкласс.

Использование ключевого слова `final` в сочетании с наследованием

Существует три способа использования ключевого слова `final`. Первый способ — его можно применять для создания эквивалента именованной константы. Это применение было описано в предыдущей главе. Остальные два применения относятся к наследованию. Давайте рассмотрим их.

Использование ключевого слова `final` для предотвращения переопределения

Хотя переопределение методов — одно из наиболее мощных средств Java, в некоторых случаях его желательно избегать. Чтобы запретить переопределение метода, в начале его объявления необходимо указать ключевое слово `final`. Методы, объявленные как `final`, переопределяться не могут. Следующий фрагмент кода иллюстрирует это применение ключевого слова `final`.

```
class A {
    final void meth() {
        System.out.println("Это метод final.");
    }
}

class B extends A {
    void meth() { // ОШИБКА! Этот метод не может быть переопределен.
        System.out.println("Не допускается!");
    }
}
```

Поскольку метод `meth()` объявлен как `final`, он не может быть переопределен в классе `B`. Попытка выполнить это переопределение приведет к ошибке времени компиляции.

Иногда методы, объявленные как `final`, могут способствовать увеличению производительности программы. Компилятор вправе *встраивать* вызовы этих методов, поскольку он "знает", что они не будут переопределены подклассом. Часто при вызове небольшого финального метода компилятор Java может встраивать код виртуальной машины подпрограммы непосредственно в скомпилированный код вызывающего метода, тем самым снижая значительные накладные расходы системных ресурсов, связанные с вызовом метода. Встраивание финальных методов в вызывающий код — лишь потенциальная возможность. Обычно Java разрешает вызовы методов динамически, во время выполнения. Такой подход называют *поздним связыванием*. Однако поскольку финальные методы не могут переопределяться, обращение к такому методу может быть разрешено во время компиляции. Этот подход называют *ранним связыванием*.

Использование ключевого слова `final` для предотвращения наследования

Иногда необходимо предотвратить наследование класса. Для этого в начале объявления класса следует поместить ключевое слово `final`. Объявление класса финальным неявно объявляет финальными и все его методы. Как легко догадаться, одновременное объявление класса как `abstract` и `final` недопустимо, поскольку абстрактный класс принципиально является незавершенным и только его подклассы предоставляют полную реализацию методов.

Ниже приведен пример финального класса.

```
final class A {
    // ...
}

// Следующий класс недопустим.
class B extends A { // ОШИБКА! Класс A не может иметь подклассы.
    // ...
}
```

Как видно из комментария, класс `B` не может происходить от класса `A`, поскольку класс `A` объявлен финальным.

Класс Object

В Java определен один специальный класс — Object. Все остальные классы являются подклассами этого класса. То есть Object — суперкласс всех остальных классов. Это означает, что ссылочная переменная класса Object может ссылаться на объект любого другого класса. Кроме того, поскольку массивы реализованы в виде классов, переменная класса Object может ссылаться также на любой массив.

Класс Object определяет методы, описанные в табл. 8.1, которые доступны в любом объекте.

Таблица 8.1. Методы класса Object

Метод	Назначение
`Object clone()`	Создает новый объект, не отличающийся от клонируемого
`boolean equals(Object object)`	Определяет, равен ли один объект другому
`void finalize()`	Вызывается перед удалением неиспользуемого объекта
`Class<?> getClass()`	Получает класс объекта во время выполнения
`int hashCode()`	Возвращает хеш-код, связанный с вызывающим объектом
`void notify()`	Возобновляет выполнение потока, который ожидает вызывающего объекта
`void notifyAll()`	Возобновляет выполнение всех потоков, которые ожидают вызывающего объекта
`String toString()`	Возвращает строку, которая описывает объект

`void wait()`
`void wait(long` *миллисекунд*`)`
`void wait(long` *миллисекунд*`,`
`int` *наносекунд*`)`
	Ожидает другого потока выполнения

Методы getClass(), notify(), notifyAll() и wait() объявлены как final. Остальные методы можно переопределять. Эти методы описаны в других главах книги. Однако обратите внимание на два метода: equals() и toString(). Метод equals() сравнивает два объекта. Если объекты равны, он возвращает значение true, если нет — false. Точное определение равенства зависит от типа сравниваемых объектов. Метод toString() возвращает строку, которая содержит описание объекта, по отношению к которому он вызван. Кроме того, этот метод автоматически вызывается при выводе объекта с помощью метода println(). Многие классы переопределяют этот метод. Это позволяет им приспосабливать описание специально для создаваемых ими объектных типов.

Последний момент: обратите внимание на необычный синтаксис в типе возвращаемого значения метода getClass(). Это имеет отношение к *обобщениям* Java, которые описываются в главе 14.

ГЛАВА 9

Пакеты и интерфейсы

В этой главе рассматриваются две наиболее новаторские концепции языка Java: пакеты и интерфейсы. *Пакеты* — это контейнеры классов. Они используются для сохранения изоляции пространства имен класса. Например, пакет позволяет создать класс по имени `List`, который можно хранить в отдельном пакете, не беспокоясь о возможных конфликтах с другим классом `List`, хранящимся в каком-то другом месте. Пакеты хранятся в иерархической структуре и явно импортируются в определения новых классов.

В предшествующих главах было описано использование методов для определения интерфейса к данным класса. С помощью ключевого слова `interface` Java позволяет полностью абстрагировать интерфейс от его реализации. Используя это ключевое слово, можно указать набор методов, которые могут быть реализованы одним или несколькими классами. В действительности сам по себе интерфейс не определяет никакой реализации. Хотя они подобны абстрактным классам, интерфейсы предоставляют дополнительную возможность: один класс может реализовать более одного интерфейса. И наоборот, класс может наследоваться только от одного суперкласса (абстрактного или не абстрактного).

Пакеты

Ранее для всех примеров классов мы использовали имена из одного пространства имен. Это означает, что во избежание конфликта имен для каждого класса нужно было указывать уникальное имя. По истечении некоторого времени при отсутствии какого-либо способа управления пространством имен может возникнуть ситуация, когда выбор удобных описательных имен отдельных классов станет затруднительным. Кроме того, требуется также какой-нибудь способ обеспечения того, чтобы выбранное имя класса было достаточно уникальным и не конфликтовало с именами классов, выбранными другими программистами. (Представьте себе небольшую группу программистов, спорящих о том, кто имеет право использовать имя "Foobar" в качестве имени класса. Или вообразите себе все сообщество Интернета, спорящее о том, кто первым назвал класс "Espresso".) К счастью, язык Java предоставляет механизм разделения пространства имен на более удобные для управления фрагменты. Этим механизмом является пакет, который одновременно используется и как механизм присвоения имен, и механизм управления видимостью.

Внутри пакета можно определить классы, не доступные коду вне этого пакета. Можно также определить члены класса, которые видны только другим членам этого же пакета. Такой механизм позволяет классам располагать полными сведениями друг о друге, но не предоставлять эти сведения остальному миру.

Определение пакета

Создание пакета является простой задачей: достаточно включить команду `package` в качестве первого оператора исходного файла Java. Любые классы, объявленные внутри этого файла, будут принадлежать указанному пакету. Оператор `package` определяет пространство имен, в котором хранятся классы. Если оператор `package` отсутствует, имена классов помещаются в используемый по умолчанию пакет без имени. (Именно поэтому до сих пор нам не нужно было беспокоиться об определении пакетов.) Хотя для коротких примеров программ пакет, используемый по умолчанию, вполне подходит, он не годится для реальных приложений. В большинстве случаев для кода придется определять пакет.

Оператор `package` имеет следующую общую форму.

```
package пакет;
```

Здесь *пакет* задает имя пакета. Например, показанный ниже оператор создает пакет `MyPackage`.

```
package MyPackage;
```

Для хранения пакетов система Java использует каталоги файловой системы. Например, файлы `.class` любых классов, объявленных в качестве составной части пакета `MyPackage`, должны храниться в каталоге `MyPackage`. Помните, что регистр символов имеет значение, а имя каталога должно в точности совпадать с именем пакета.

Один и тот же оператор `package` может присутствовать в более чем одном файле. Этот оператор просто указывает пакет, к которому принадлежат классы, определенные в данном файле. Он не препятствует тому, чтобы иные классы в других файлах были частью этого же пакета. Большинство пакетов, используемых в реальных программах, распределено по множеству файлов.

Язык Java позволяет создавать иерархию пакетов. Для этого применяется точечный оператор. Оператор многоуровневого пакета имеет следующую общую форму.

```
package пакет1[.пакет2[.пакет3]];
```

Иерархия пакетов должна быть отражена в файловой системе среды разработки Java. Например, в среде Windows пакет, объявленный как `package java.awt.image;`, должен храниться в каталоге `java\awt\image`. Необходимо тщательно проверять правильность выбора имен пакетов. Имя пакета нельзя изменить, не изменяя имя каталога, в котором хранятся классы.

Поиск пакетов и переменная среды CLASSPATH

Как было сказано в предыдущем разделе, пакеты соответствуют каталогам. Это обстоятельство порождает важный вопрос: откуда системе времени выполнения Java известно, где следует искать создаваемые пакеты? Ответ на него состоит из следующих частей: во-первых, по умолчанию в качестве отправной точки система времени выполнения Java использует текущий рабочий каталог. Следовательно, если пакет находится в подкаталоге текущего каталога, он будет найден. Во-вторых, путь или пути к каталогу можно указать, устанавливая значение переменной среды `CLASSPATH`. В-третьих, вызов `java` и `javac` в командной сроке можно использовать с параметром `-classpath`, указывающим путь к классам.

Например, рассмотрим следующую спецификацию пакета.

```
package MyPack;
```

Чтобы программа могла найти пакет MyPack, должно выполняться одно из следующих двух условий. Либо программа должна выполняться из каталога, расположенного непосредственно над каталогом MyPack, либо переменная среды CLASSPATH должна содержать путь к каталогу MyPack, либо параметр -classpath должен указывать путь к каталогу MyPack во время запуска программы java.

При использовании двух последних способов путь класса *не должен содержать* сам пакет MyPack. Он должен просто указывать *путь* к этому каталогу. Например, в среде Windows, если путь к каталогу MyPack имеет вид C:\MyPrograms\Java\MyPack, путь класса к пакету MyPack будет выглядеть так.

```
C:\MyPrograms\Java
```

Простейший способ проверки примеров, приведенных в этой книге, — создание каталогов пакетов в текущем каталоге разработки, помещение файлов .class в соответствующие каталоги и последующий запуск программ из каталога разработки. В следующем примере использован именно этот подход.

Краткий пример пакета

С учетом описанного выше можете попытаться использовать следующий 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();
    }
}
```

Назовите этот файл AccountBalance.java и поместите его в каталог MyPack.

Затем выполните компиляцию файла. Убедитесь, что результирующий файл .class также помещен в каталог MyPack. Затем попробуйте выполнить класс AccountBalance, вводя следующую командную строку.

```
java MyPack.AccountBalance
```

Помните, что при выполнении этой команды текущим должен быть каталог, расположенный над каталогом `MyPack`, либо переменная среды `CLASSPATH` должна содержать соответствующий путь.

Как мы уже поясняли, теперь класс `AccountBalance` — часть пакета `MyPack`. Это означает, что его нельзя выполнять самостоятельно. То есть нельзя использовать следующую командную строку.

```
java AccountBalance
```

Имя `AccountBalance` требует уточнения именем его пакета.

Защита доступа

В предыдущих главах вы рассмотрели различные аспекты механизма управления доступом Java и его модификаторы. Например, вы уже знаете, что доступ к закрытому члену класса предоставляется только другим членам этого класса. Пакеты добавляют к управлению доступом еще одно измерение. Как вы вскоре убедитесь, Java предоставляет множество уровней защиты, обеспечивая очень точное управление видимостью переменных и методов внутри классов, подклассов и пакетов.

Классы и пакеты одновременно служат средствами инкапсуляции и хранилищем пространства имен и области видимости переменных и методов. Пакеты играют роль контейнеров классов и других подчиненных пакетов. Классы служат контейнерами данных и кода. Класс — наименьшая единица абстракции Java. Вследствие взаимодействия между классами и пакетами Java определяет четыре категории видимости членов класса.

- Подклассы в одном пакете.
- Классы в одном пакете, не являющиеся подклассами.
- Подклассы в различных пакетах.
- Классы, которые не находятся в одном пакете и не являются подклассами.

Три модификатора доступа — `private`, `public` и `protected` — предоставляют разнообразные способы создания множества уровней доступа, необходимых для этих категорий. Взаимосвязь между ними описана в табл. 9.1.

Таблица 9.1. Доступ к членам класса

	Private	Модификатор отсутствует	Protected	Public
Один и тот же класс	Да	Да	Да	Да
Подкласс класса этого же пакета	Нет	Да	Да	Да
Класс этого же пакета, не являющийся подклассом	Нет	Да	Да	Да
Подкласс класса другого пакета	Нет	Нет	Да	Да
Класс другого пакета, не являющийся подклассом класса данного пакета	Нет	Нет	Нет	Да

Хотя на первый взгляд механизм управления доступом Java может показаться сложным, следующие соображения могут облегчить его понимание. Любой компонент, объявленный как `public`, доступен из любого кода. Любой компонент, объявленный как `private`, не виден для компонентов, расположенных вне его класса. Если член не содержит явного модификатора доступа, он видим подклассам и другим классам в данном пакете. Этот уровень доступа используется по умол-

чанию. Если нужно, чтобы элемент был виден за пределами его текущего пакета, но только классам, которые являются непосредственными подклассами данного класса, элемент должен быть объявлен как `protected`.

Правила доступа, описанные в табл. 9.1, применимы только к членам класса. Для класса, не являющегося вложенным, может быть указан только один из двух возможных уровней доступа: заданный по умолчанию и `public`. Когда класс объявлен как `public`, он доступен любому другому коду. Если для класса указан уровень доступа, определенный по умолчанию, он доступен только для кода внутри данного пакета. Когда класс является открытым, он должен быть единственным открытым классом, объявленным в файле, и имя файла должно совпадать с именем класса.

Пример защиты доступа

Следующий пример демонстрирует использование всех комбинаций модификаторов управления доступом. Он содержит два пакета и пять классов. Не забудьте, что классы двух различных пакетов должны храниться в каталогах, имена которых совпадают с именами соответствующих пакетов, — в данном случае `p1` и `p2`.

Исходный файл первого пакета определяет три класса `Protection`, `Derived` и `SamePackage`. Первый класс определяет четыре переменные типа `int` — по одной в каждом из допустимых режимов защиты доступа. Переменная `n` объявлена с уровнем защиты, используемым по умолчанию, `n_pri` — как `private`, `n_pro` — `protected`, а `n_pub` — `public`.

В этом примере все другие классы будут предпринимать попытку обращения к переменным экземпляра этого класса. Строки, компиляция которых невозможна из-за нарушений правил доступа, оформлены в виде комментариев. Перед каждой из этих строк помещен комментарий с указанием точек программы, из которых был бы возможен доступ к этому уровню защиты.

Второй класс, `Derived`, — подкласс класса `Protection` этого же пакета `p1`. Он предоставляет классу `Derived` доступ ко всем переменным класса `Protection`, кроме переменной `n_pri`, объявленной как `private`. Третий класс, `SamePackage`, не является подклассом класса `Protection`, но находится в этом же пакете и обладает доступом ко всем переменным, кроме переменной `n_pri`.

Файл `Protection.java` содержит следующий код.

```
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);
    }
}
```

Файл `Derived.java` содержит такой код.

```
package p1;

class Derived extends Protection {
```

```
    Derived() {
        System.out.println("конструктор подкласса");
        System.out.println("n = " + n);

     // доступно только для класса
     // System.out.println("n_pri = "4 + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

Файл `SamePackage.java` содержит следующий код.

```
package p1;

class SamePackage {
    SamePackage() {

        Protection p = new Protection();
        System.out.println("конструктор этого же пакета");
        System.out.println("n = " + p.n);

     // class only
     // 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 приведен исходный код второго пакета, `p2`. Два определенных в нем класса отражают оставшиеся две ситуации управления доступом. Первый класс, `Protection2`, — это подкласс класса `p1.Protection`. Он имеет доступ ко всем переменным класса `p1.Protection`, кроме переменных `n_pri` (поскольку она объявлена как `private`) и `n`, которая объявлена с уровнем защиты, используемым по умолчанию. Вспомните, что заданный по умолчанию режим доступа разрешает доступ из данного класса или пакета, но не из подклассов другого пакета. И наконец, класс `OtherPackage` имеет доступ только к одной переменной — `n_pub`, которая была объявлена как `public`.

Файл `Protection2.java` содержит следующий код.

```
package p2;

class Protection2 extends p1.Protection {
    Protection2() {
        System.out.println("унаследованный конструктор другого пакета");

     // доступно только для данного класса или пакета
     // System.out.println("n = " + n);

     // доступно только для данного класса
     // System.out.println("n_pri = " + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

Файл `OtherPackage.java` содержит следующий код.

```
package p2;

class OtherPackage {
    OtherPackage() {
        p1.Protection p = new p1.Protection();
        System.out.println("конструктор другого пакета");

     // доступно только для данного класса или пакета
     // System.out.println("n = " + p.n);

     // доступно только для данного класса
     // System.out.println("n_pri = " + p.n_pri);

     // доступно только для данного класса, подкласса или пакета
     // System.out.println("n_pro = " + p.n_pro);

        System.out.println("n_pub = " + p.n_pub);
    }
}
```

Для проверки работы этих двух пакетов можно использовать следующие два проверочных файла. Проверочный файл для пакета `p1` имеет такой вид.

```
// Демонстрационный пакет p1.
package p1;

// Конкретизация различных классов пакета p1.
public class Demo {
    public static void main(String args[]) {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
    }
}
```

Следующий файл — проверочный файл пакета `p2`.

```
// Демонстрационный пакет p2.
package p2;

// Конкретизация различных классов пакета p2.
public class Demo {
    public static void main(String args[]) {
        Protection2 ob1 = new Protection2();
        OtherPackage ob2 = new OtherPackage();
    }
}
```

Импорт пакетов

Если вспомнить, что пакеты предлагают эффективный механизм изоляции различных классов друг от друга, становится понятно, почему все встроенные классы Java хранятся в пакетах. Ни один из основных классов Java не хранится в неименованном пакете, используемом по умолчанию. Все стандартные классы хранятся в каком-либо именованном пакете. Поскольку внутри пакетов классы должны быть полностью определены именами их пакетов, длинное, разделенное точками имя пути пакета каждого используемого класса может оказаться слишком громоздким. Поэтому, чтобы определенные классы или весь пакет можно было сделать видимыми, в Java включен оператор `import`. После того как класс импортирован,

на него можно ссылаться непосредственно, используя только его имя. Оператор `import` служит только для удобства программистов и не является обязательным с технической точки зрения для создания завершенной программы Java. Однако если в приложении придется ссылаться на несколько десятков классов, оператор `import` значительно уменьшит объем вводимого кода.

В исходном файле программы Java операторы `import` должны следовать непосредственно за оператором `package` (если таковой имеется) перед любыми определениями классов. Оператор `import` имеет следующую общую форму.

```
import пакет1 [.пакет2].(имя_класса | *);
```

В этой форме *пакет1* — имя пакета верхнего уровня, *пакет2* — имя подчиненного пакета внутри внешнего пакета, отделенное символом "точка" (`.`). Глубина вложенности пакетов практически не ограничена ничем, кроме файловой системы. И наконец, *имя_класса* может быть задано либо явно, либо с помощью символа "звездочка" (`*`), который указывает компилятору Java о необходимости импорта всего пакета. Следующий фрагмент демонстрирует применение обеих форм оператора.

```
import java.util.Date;
import java.io.*;
```

Все стандартные классы, поставляемые с системой Java, хранятся в пакете `java`. Основные функции языка хранятся в пакете `java.lang` внутри пакета `java`. Обычно каждый пакет или класс, который нужно использовать, приходится импортировать. Но поскольку система Java бесполезна без многих функций, определенных в пакете `java.lang`, компилятор неявно импортирует его для всех программ. Это эквивалентно наличию следующей строки в каждой из программ.

```
import java.lang.*;
```

При наличии в двух различных пакетах, импортируемых с применением формы со звездочкой, классов с одинаковыми именами компилятор никак на это не отреагирует, если только не будет предпринята попытка использования одного из этих классов. В этом случае возникнет ошибка времени компиляции, и имя класса придется указать явно, задавая его пакет.

Полностью определенное имя класса с указанием полной иерархии пакетов можно использовать везде, где допускается имя класса. Например, в следующем фрагменте кода присутствует оператор импорта.

```
import java.util.*;
class MyDate extends Date {
}
```

Этот же пример без оператора `import` выглядит следующим образом.

```
class MyDate extends java.util.Date {
}
```

В этой версии объект `Date` полностью определен.

Как видно в табл. 9.1, при импорте пакета в импортирующем коде классам, не являющимся подклассами классов пакета, будут доступны только те элементы пакета, которые объявлены как `public`. Например, если нужно, чтобы приведенный ранее класс `Balance` пакета `MyPack` был доступен в качестве самостоятельного класса вне пакета `MyPack`, его необходимо объявить как `public` и поместить в отдельный файл, как показано в следующем примере.

```
package MyPack;

/* Теперь класс Balance, его конструктор и его метод show()
```

```
   являются открытыми. Это означает, что вне их пакета они
   могут использоваться кодом, не являющимся подклассом 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);
    }
}
```

Как видите, теперь класс `Balance` объявлен как `public`. Его конструктор и метод `show()` также объявлены как `public`. Это означает, что они доступны любому коду вне пакета `MyPack`. Например, класс `TestBalance` импортирует пакет `MyPack` и поэтому может использовать класс `Balance`.

```
import MyPack.*;

class TestBalance {
    public static void main(String args[]) {
        /* Поскольку класс Balance объявлен как public, его можно
           использовать и вызывать его конструктор. */
        Balance test = new Balance("J. J. Jaspers", 99.88);

        test.show(); // можно также вызывать метод show()
    }
}
```

В качестве эксперимента удалите модификатор `public` из класса `Balance`, а затем попытайтесь выполнить компиляцию класса `TestBalance`. Как уже было сказано, это приведет к возникновению ошибок.

Интерфейсы

Применение ключевого слова `interface` позволяет полностью абстрагировать интерфейс класса от его реализации. То есть с использованием ключевого слова `interface` можно задать действия, которые должен выполнять класс, но не то, как именно он должен это делать. Синтаксически интерфейсы аналогичны классам, но не содержат переменных экземпляров, а объявления их методов не содержат тела метода. На практике это означает, что можно объявлять интерфейсы, которые не делают никаких допущений относительно их реализации. Как только интерфейс определен, его может реализовать любое количество классов. Кроме того, один класс может реализовать любое количество интерфейсов.

Чтобы реализовать интерфейс, класс должен создать полный набор методов, определенных интерфейсом. Однако каждый класс может определять нюансы своей реализации данного интерфейса. Ключевое слово `interface` позволяет в полной мере использовать концепцию полиморфизма под названием "один интерфейс, несколько методов".

Интерфейсы предназначены для поддержки динамического разрешения методов во время выполнения. Обычно, чтобы вызов метода мог выполняться из одного класса в другом, оба класса должны присутствовать во время компиляции, дабы компилятор Java мог проверить совместимость сигнатур методов. Само по себе это требование создает статическую и нерасширяемую среду обработки классов. В такой системе функциональные возможности неизбежно передаются по иерархии классов все выше и выше, в результате чего механизмы будут становиться доступными все большему количеству подклассов. Интерфейсы предназначены для предотвращения этой проблемы. Они изолируют определение метода или набора методов от иерархии наследования. Поскольку иерархия интерфейсов не совпадает с иерархией классов, классы, никак не связанные между собой в иерархии классов, могут реализовать один и тот же интерфейс. Именно здесь возможности интерфейсов проявляются наиболее полно.

На заметку! Интерфейсы добавляют большинство функциональных возможностей, требуемых многим приложениям, которым в обычных условиях в языках вроде C++ пришлось бы прибегать к использованию множественного наследования.

Определение интерфейса

Во многом определение интерфейса подобно определению класса. Упрощенная общая форма интерфейса имеет следующий вид.

```
доступ interface имя {
возвращаемый_тип имя_метода1(список_параметров);
возвращаемый_тип имя_метода2(список_параметров);
тип имя_конечной_переменной1 = значение;
тип имя_конечной_переменной2 = значение;
// ...
возвращаемый_тип имяМетодаN(список_параметров);
тип имя_конечной_переменнойN = значение;
}
```

Если определение не содержит никакого модификатора доступа, используется доступ по умолчанию и интерфейс доступен только другим членам того пакета, в котором он объявлен. Если интерфейс объявлен как `public`, он может быть использован любым другим кодом. В этом случае интерфейс должен быть единственным открытым интерфейсом, объявленным в файле, и имя файла должно совпадать с именем интерфейса. *Имя* — имя интерфейса, которым может быть любой допустимый идентификатор. Обратите внимание на то, что объявляемые методы не содержат тел. Их объявления завершаются списком параметров, за которым следует символ "точка с запятой". По сути, они представляют собой абстрактные методы. Ни один из указанных внутри интерфейса методов не может обладать никакой заданной по умолчанию реализацией. Каждый класс, который включает в себя интерфейс, должен реализовать все его методы.

Переменные могут быть объявлены внутри объявлений интерфейсов. Они неявно объявляются как `final` и `static`, т.е. реализующий класс не может их изменять. Кроме того, они должны быть также инициализированы. Все методы и переменные неявно объявляются как `public`.

Ниже приведен пример определения интерфейса. В нем объявляется простой интерфейс, который содержит один метод `callback()`, принимающий единственный целочисленный параметр.

```
interface Callback {
    void callback(int param);
}
```

Реализация интерфейсов

Как только интерфейс определен, его может реализовать один или несколько классов. Чтобы реализовать интерфейс, в определение класса потребуется включить конструкцию `implements`, а затем создать методы, определенные интерфейсом. Общая форма класса, который содержит выражение `implements`, имеет следующий вид.

```
доступ class имя_класса [extends суперкласс]
          [implements интерфейс [,интерфейс...]] {
    // тело_класса
}
```

Если класс реализует более одного интерфейса, имена интерфейсов разделяются запятыми. Если класс реализует два интерфейса, которые объявляют один и тот же метод, то один и тот же метод будет использоваться клиентами любого интерфейса. Методы, которые реализуют интерфейс, должны быть объявлены как `public`. Кроме того, сигнатура типа реализующего метода должна в точности совпадать с сигнатурой типа, указанной в определении `interface`.

Рассмотрим небольшой пример класса, который реализует приведенный ранее интерфейс `Callback`.

```
class Client implements Callback {
    // Реализует интерфейс Callback
    public void callback(int p) {

        System.out.println("Метод callback, вызванный со значением " +
                           p);
    }
}
```

Обратите внимание на то, что метод `callback()` объявлен с использованием модификатора доступа `public`.

Помните! При реализации метода интерфейса он должен быть объявлен как `public`.

Вполне допустима и достаточно распространена ситуация, когда классы, которые реализуют интерфейсы, определяют собственные дополнительные члены. Например, следующая версия класса `Client` реализует метод `callback()` и добавляет метод `nonInfaceMeth()`.

```
class Client implements Callback {
    // Реализует интерфейс Callback
    public void callback(int p) {
        System.out.println("Метод callback, вызванный со значением " +
                           p);
    }

    void nonIfaceMeth() {
        System.out.println("Классы, которые реализуют интерфейсы" +
                           "могут определять также и другие члены.");
    }
}
```

Доступ к реализациям через ссылки на интерфейсы

Переменные можно объявлять как объектные ссылки, которые используют тип интерфейса, а не тип класса. При помощи такой переменной можно ссылаться на

любой экземпляр любого класса, реализующего объявленный интерфейс. При вызове метода с помощью одной из таких ссылок выбор нужной версии будет производиться в зависимости от конкретного экземпляра интерфейса, на который выполняется ссылка. Это — одна из главных особенностей интерфейсов. Поиск выполняемого метода осуществляется динамически во время выполнения, что позволяет создавать классы позже, чем код, который вызывает методы по отношению к этим классам. Диспетчеризация кода может выполняться с использованием интерфейса без необходимости наличия каких-либо сведений о "вызывающем". Этот процесс аналогичен использованию ссылки на суперкласс для доступа к объекту подкласса, описанному в главе 8.

Внимание! Поскольку в системе Java динамический поиск методов во время выполнения сопряжен со значительными накладными расходами по сравнению с обычным вызовом методов, в коде, для которого важна производительность, интерфейсы следует использовать только тогда, когда это действительно необходимо.

В следующем примере метод `callback()` вызывается через ссылочную переменную интерфейса.

```
class TestIface {
    public static void main(String args[]) {
        Callback c = new Client();
        c.callback(42);
    }
}
```

Эта программа создает следующий вывод.

```
Метод callback, вызванный со значением 42
```

Обратите внимание на то, что хотя переменная `c` объявлена с типом интерфейса `Callback`, ей был присвоен экземпляр класса `Client`. Хотя переменную `c` можно использовать для доступа к методу `callback()`, она не имеет доступа к каким-то другим членам класса `Client`. Ссылочная переменная интерфейса располагает только сведениями о тех методах, которые объявлены в ее объявлении `interface`. Таким образом, переменная `c` не может применяться для доступа к методу `nonIfaceMeth()`, поскольку она объявлена классом `Client`, а не классом `Callback`.

Хотя приведенный пример формально показывает, как ссылочная переменная интерфейса может получать доступ к объекту реализации, он не демонстрирует полиморфные возможности такой ссылки. Чтобы продемонстрировать пример такого применения, вначале создадим вторую реализацию интерфейса `Callback`.

```
// Еще одна реализация интерфейса Callback.
class AnotherClient implements Callback {
    // Реализация интерфейса Callback
    public void callback(int p) {
        System.out.println("Еще одна версия callback");
        System.out.println("p в квадрате равно " + (p*p));
    }
}
```

Теперь проверим работу следующего класса.

```
class TestIface2 {
    public static void main(String args[]) {
        Callback c = new Client();
        AnotherClient ob = new AnotherClient();

        c.callback(42);
```

```
        c = ob; // теперь с ссылается на объект AnotherClient
        c.callback(42);
    }
}
```

Эта программа создает следующий вывод.

```
callback вызванный со значением 42
Еще одна версия callback
р в квадрате равно 1764
```

Как видите, вызываемая версия метода `callback()` определяется типом объекта, на который переменная с ссылается во время выполнения. Представленный пример очень прост, поэтому вскоре мы приведем еще один, более реальный пример.

Частичные реализации

Если класс содержит интерфейс, но не полностью реализует определенные им методы, он должен быть объявлен как `abstract` (абстрактный).

```
abstract class Incomplete implements Callback {
    int a, b;

    void show() {
        System.out.println(a + " " + b);
    }
    // ...
}
```

В этом примере класс `Incomplete` не реализует метод `callback()` и должен быть объявлен как абстрактный. Любой класс, который наследует класс `Incomplete`, должен реализовать метод `callback()` либо быть также объявлен как `abstract`.

Вложенные интерфейсы

Интерфейс может быть объявлен членом класса или другого интерфейса. Такой интерфейс называется *интерфейсом-членом* или *вложенным интерфейсом*. Вложенный интерфейс может быть объявлен как `public`, `private` или `protected`. Это отличает его от интерфейса верхнего уровня, который должен быть либо объявлен как `public`, либо, как уже было отмечено, должен использовать уровень доступа, заданный по умолчанию. Когда вложенный интерфейс используется вне содержащей его области видимости, он должен определяться именем класса или интерфейса, членом которого является. То есть вне класса или интерфейса, в котором объявлен вложенный интерфейс, его имя должно быть полностью определено.

В следующем примере демонстрируется применение вложенного интерфейса.

```
// Пример вложенного интерфейса.

// Этот класс содержит интерфейс-член.
class A {
    // это 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[]) {

    // использует 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 не будет отображаться");
    }
}
```

Обратите внимание на то, что объект `A` определяет вложенный интерфейс `NestedIF`, который объявлен как `public`. Затем объект `B` реализует вложенный интерфейс, указав следующее.

```
implements A.NestedIF
```

Обратите также внимание на то, что имя интерфейса полностью определено и содержит имя класса. Внутри метода `main()` создается ссылка на интерфейс `A.NestedIF` с именем `nif`, которой присваивается ссылка на объект `B`. Поскольку объект `B` реализует интерфейс `A.NestedIF`, это допустимо.

Использование интерфейсов

Чтобы возможности интерфейсов были понятны, рассмотрим более реальный пример. В предыдущих главах мы разработали класс `Stack`, который реализует простой стек фиксированного размера. Однако существует множество способов реализации стека. Например, стек может иметь фиксированный размер либо быть "увеличивающимся". Стек может также храниться в массиве, связанном списке, бинарном дереве и т.п. Независимо от реализации стека, его интерфейс остается неизменным. То есть методы `push()` и `pop()` определяют интерфейс стека независимо от нюансов реализации. Поскольку интерфейс стека отделен от его реализации, можно без труда определить интерфейс стека, предоставляя реализации определение специфичных особенностей. Рассмотрим два примера.

Вначале создадим интерфейс, который определяет целочисленный стек. Поместим его в файл `IntStack.java`. Этот интерфейс будет использоваться обеими реализациями стека.

```
// Определение интерфейса целочисленного стека.
interface IntStack {
    void push(int item); // сохранение элемента
    int pop();           // извлечение элемента
}
```

Следующая программа создает класс `FixedStack`, который реализует версию целочисленного стека фиксированной длины.

```
// Реализация IntStack, использующая область хранения
// фиксированного размера.
class FixedStack implements IntStack {
    private int stck[];
```

```
    private int tos;
    // резервирование и инициализация стека
    FixedStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // заталкивание элемента в стек
    public void push(int item) {
        if(tos==stck.length-1) // использование члена длины стека
            System.out.println("Стек полон.");
        else
        stck[++tos] = item;
    }

    // выталкивание элемента из стека
    public int pop() {
        if(tos < 0) {
            System.out.println("Стек пуст.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class IFTest {
public static void main(String args[]) {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);

    // заталкивание чисел в стек
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // выталкивание этих чисел из стека
    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());
    }
}
```

Теперь создадим еще одну реализацию интерфейса `IntStack`, которая, используя то же самое определение `interface`, создает динамический стек. В этой реализации каждый стек создается с начальной длиной. При превышении этой начальной длины размер стека увеличивается. Каждый раз, когда возникает потребность в дополнительном месте, размер стека удваивается.

```
// Реализация "увеличивающегося" стека.
class DynStack implements IntStack {
    private int stck[];
    private int tos;

    // резервирование и инициализация стека
    DynStack(int size) {
        stck = new int[size];
        tos = -1;
```

```
    }

    // Заталкивание элемента в стек
    public void push(int item) {
        // если стек полон, резервирование стека большего размера
        if(tos==stck.length-1) {
            int temp[] = new int[stck.length * 2]; // удвоение размера
            for(int i=0; i<stck.length; i++) temp[i] = stck[i];
            stck = temp;
            stck[++tos] = item;
        }
        else
            stck[++tos] = item;
    }

    // Выталкивание элемента из стека
    public int pop() {
        if(tos < 0) {
            System.out.println("Стек пуст.");
            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 увеличивают размеры каждого из стеков
        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());
    }
}
```

Следующий класс использует обе реализации классов `FixedStack` и `DynStack`. Для этого применяется ссылка на интерфейс. Это означает, что поиск версий при обращении к методам `push()` и `pop()` осуществляется во время выполнения, а не во время компиляции.

```
/* Создание переменной интерфейса и
   обращение к стекам через нее.
*/
class IFTest3 {
    public static void main(String args[]) {
        IntStack mystack; // создание ссылочной переменной интерфейса
        DynStack ds = new DynStack(5);
        FixedStack fs = new FixedStack(8);

        mystack = ds;     // загрузка динамического стека
        // заталкивание чисел в стек
```

```
        for(int i=0; i<12; i++) mystack.push(i);

        mystack = fs;      // загрузка фиксированного стека
        for(int i=0; i<8; i++) mystack.push(i);

        mystack = ds;
        System.out.println("Значения в динамическом стеке:");
        for(int i=0; i<12; i++)
            System.out.println(mystack.pop());

        mystack = fs;
        System.out.println("Значения в фиксированном стеке:");
        for(int i=0; i<8; i++)
            System.out.println(mystack.pop());
    }
}
```

В этой программе `mystack` – ссылка на интерфейс `IntStack`. Таким образом, когда она ссылается на переменную `ds`, программа использует версии методов `push()` и `pop()`, определенные реализацией `DynStack`. Когда же она ссылается на переменную `fs`, программа использует версии методов `push()` и `pop()`, определенные реализацией `FixedStack`. Как уже было сказано, эти решения принимаются во время выполнения. Обращение к нескольким реализациям интерфейса через ссылочную переменную интерфейса – наиболее мощный метод поддержки полиморфизма времени выполнения Java.

Переменные в интерфейсах

Интерфейсы можно применять для импорта совместно используемых констант в несколько классов за счет простого объявления интерфейса, который содержит переменные, инициализированные нужными значениями. При включении интерфейса в класс (т.е. при "реализации" интерфейса) имена всех этих переменных будут помещены в область констант. (Это аналогично использованию в программе C/C++ заголовочного файла для создания большого количества констант типа `#define` или объявлений `const`.) Если интерфейс не содержит никаких методов, любой класс, который включает в себя такой интерфейс, в действительности ничего не реализует. Это равносильно тому, что класс импортировал бы постоянные поля в пространство имен класса в качестве финальных переменных. В следующем примере эта технология 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("Позднее");
                break;
            case SOON:
                System.out.println("Вскоре");
                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 на то, что в этой программе использован один из стандартных классов Java — `Random`. Этот класс создает псевдослучайные числа. Он содержит несколько методов, которые позволяют получать случайные числа в требуемой программой форме. В этом примере применяется метод `nextDouble()`, который возвращает 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 константы `NO` (Нет), `YES` (Да), `MAYBE` (Возможно), `SOON` (Вскоре), `LATER` (Позднее) и `NEVER` (Никогда). Код внутри каждого класса ссылается на эти константы так, как если бы каждый класс определял или наследовал их непосредственно. Ниже показан вывод, полученный в результате выполнения этой программы. Обратите внимание на то, что при каждом запуске результаты выполнения программы будут различными.

```
Позднее
Вскоре
Нет
Да
```

Возможность расширения интерфейсов

Ключевое слово `extends` позволяет одному интерфейсу наследовать другой. Синтаксис определения такого наследования аналогичен синтаксису наследования классов. Когда класс реализует интерфейс, который наследует другой интерфейс, он должен предоставлять реализации всех методов, определенных внутри цепочки наследования интерфейса. Ниже показан пример.

```
// Один интерфейс может расширять другой.
interface A {
    void meth1();
    void meth2();
}

// Теперь B включает в себя meth1() и meth2() и добавляет meth3().
interface B extends A {
    void meth3();
}

// Этот класс должен реализовать все методы классов 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 arg[]) {
        MyClass ob = new MyClass();

        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}
```

В порядке эксперимента можете попытаться удалить реализацию метода `meth1()` из класса `MyClass`. Это приведет к ошибке времени компиляции. Как уже было сказано, любой класс, который реализует интерфейс, должен реализовать все определенные этим интерфейсом методы, в том числе любые методы, унаследованные от других интерфейсов.

Хотя в приведенных в этой книге примерах пакеты или интерфейсы используются не очень часто, оба средства являются важными составляющими среды программирования Java. Буквально все реальные программы, написанные на языке Java, будут храниться в пакетах. Вполне вероятно, что многие из них будут также реализовать интерфейсы. Поэтому важно освоить их применение.

ГЛАВА 10 Обработка исключений

В этой главе рассматривается механизм обработки исключений Java. *Исключение* – это нештатная ситуация, возникающая во время выполнения последовательности кода. Другими словами, исключение – это ошибка времени выполнения. В языках программирования, которые не поддерживают обработки исключений, ошибки должны проверяться и обрабатываться "вручную" – как правило, за счет использования кодов ошибок и т.п. Этот подход как обременителен, так и чреват проблемами. Обработка исключений Java позволяет избежать этих проблем и, кроме того, переносит управление ошибками времени выполнения в объектно-ориентированный мир.

Основы обработки исключений

Исключение Java представляет собой объект, который описывает исключительную (т.е. ошибочную) ситуацию, возникающую в части программного кода. Когда возникает такая ситуация, в вызвавшем ошибку методе *создается и передается* объект, который представляет исключение. Этот метод может либо обработать исключение самостоятельно, либо пропустить его. В обоих случаях в некоторой точке исключение *перехватывается* и *обрабатывается*. Исключения могут создаваться системой времени выполнения Java либо могут быть созданы вручную вашим кодом. Исключения, которые передает Java, имеют отношение к фундаментальным ошибкам, которые нарушают правила языка Java либо ограничения системы выполнения Java. Исключения, созданные вручную, обычно применяются для сообщения о неких ошибках вызывающей стороне метода.

Обработка исключений Java управляется пятью ключевыми словами: `try`, `catch`, `throw`, `throws` и `finally`. Если кратко, они работают следующим образом. Операторы программы, которые вы хотите отслеживать на предмет исключений, помещаются в блок `try`. Если исключение происходит в блоке `try`, оно создается и передается. Ваш код может перехватить исключение (используя блок `catch`) и обработать его некоторым осмысленным способом. Системные исключения автоматически передаются системой времени выполнения Java. Чтобы передать исключение вручную, используется ключевое слово `throw`. Любое исключение, которое создается и передается внутри метода, должно быть указано в его интерфейсе ключевым словом `throws`. Любой код, который в обязательном порядке должен быть выполнен после завершения блока `try`, помещается в блок `finally`. Ниже показана общая форма блока обработки исключений.

```
try {
    // блок кода, в котором отслеживаются ошибки
}
```

```
catch (тип_исключения_1 exOb) {
    // обработчик исключений типа тип_исключения_1
}
catch (тип_исключения_2 exOb) {
    // обработчик исключений типа тип_исключения_2
}
// ...
finally {
    // блок кода, который должен быть выполнен
    // после завершения блока try

}
```

Здесь *тип_исключения* — тип происходящего исключения. Последующий материал настоящей главы посвящен описанию применения этой программной структуры.

На заметку! В комплекте JDK 7 добавлена новая форма оператора `try`, обеспечивающего автоматическое управление ресурсами. Эта форма, называемая `try`-*с-ресурсами*, описана в главе 13 в контексте управления файлами, поскольку файлы — это один из наиболее часто используемых ресурсов.

Типы исключений

Все типы исключений являются подклассами встроенного класса `Trowable`. То есть класс `Trowable` расположен на вершине иерархии классов исключений. Непосредственно под классом `Trowable` в ней находятся два подкласса, которые делят все исключения на две отдельные ветви. Одну ветвь возглавляет класс `Exception`. Этот класс используется для исключительных условий, которые должна перехватывать пользовательская программа. Это также класс, от которого вы будете наследовать свои подклассы при создании собственных типов исключений. У класса `Exception` имеется важный подкласс по имени `RuntimeException`. Исключения этого типа автоматически определяются для программ, которые вы пишете, и включают такие ошибки, как деление на нуль и ошибочная индексация массивов.

Другая ветвь начинается с класса `Error`, определяющего исключения, возникновение которых не ожидается при нормальном выполнении программы. Исключения типа `Error` используются системой времени выполнения Java для обозначения ошибок, происходящих внутри самой среды. Примером такой ошибки может служить переполнение стека. В этой главе не рассматриваются исключения типа `Error`, поскольку они обычно создаются в ответ на катастрофические сбои, которые не могут быть обработаны вашей программой.

Необработанные исключения

Прежде чем вы узнаете, как обрабатывать исключения в своей программе, полезно будет посмотреть, что происходит, когда вы не обрабатываете их. Следующая небольшая программа представляет пример, который намеренно вызывает ошибку деления на нуль.

```
class Exc0 {
    public static void main(String args[]) {
        int d = 0;
        int a = 42 / d;
    }
}
```

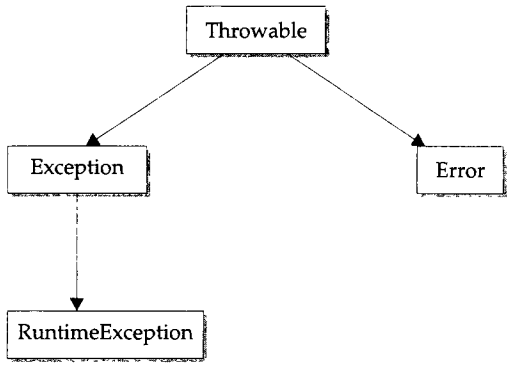


Рис. 10.1. Высокоуровневое представление иерархии исключений

Когда система времени выполнения Java обнаруживает попытку деления на нуль, она создает новый объект исключения, а затем *передает* его. Это прерывает выполнение класса `Exc0`, поскольку, как только исключение передано, оно должно быть *перехвачено* обработчиком исключений, который должен немедленно с ним что-то сделать. В данном примере мы не применили никакого собственного обработчика исключений, поэтому исключение перехватывается стандартным обработчиком, предоставленным системой времени выполнения Java. Любое исключение, которое не перехвачено вашей программой, в конечном итоге будет перехвачено и обработано этим стандартным обработчиком. Стандартный обработчик отображает строку, описывающую исключение, выводит трассировку стека от точки возникновения исключения и прерывает программу. Ниже приведен пример исключения, созданного представленным выше кодом.

```
java.lang.ArithmeticException: / by zero
    at Exc0.main(Exc0.java:4)
```

Обратите внимание на то, что имя класса `Exc0`, имя метода `main`, имя файла `Exc0.java` и номер строки `4` включены в трассировку стека. Также нужно обратить внимание на то, что переданное исключение является подклассом класса `Exception`, по имени `ArithmeticException`, который более точно описывает тип возникшей ошибки. Как будет показано далее в настоящей главе, Java применяет несколько встроенных типов исключений, соответствующих разным типам ошибок времени выполнения, которые могут быть созданы.

Трассировка стека всегда покажет последовательность вызовов методов, которая привела к ошибке. Например, вот другая версия предыдущей программы, представляющая ту же ошибку, но в методе, отдельном от метода `main()`.

```
class Exc1 {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String args[]) {
        Exc1.subroutine();
    }
}
```

Результирующая трассировка стека стандартного обработчика исключений показывает весь стек вызовов.

```
java.lang.ArithmeticException: / by zero
    at Exc1.subroutine(Exc1.java:4)
    at Exc1.main(Exc1.java:7)
```

Как видите, в нижней части стека находится строка 7 метода `main()`, в которой расположен вызов метода `subroutine()`, породивший исключение в строке 4. Трассировка стека достаточно удобна для отладки, поскольку показывает всю последовательность вызовов, приведших к ошибке.

Использование блоков `try` и `catch`

Хотя стандартный обработчик исключений, который предоставляет система времени выполнения Java, удобен для отладки, обычно вы захотите обрабатывать исключения самостоятельно. Это дает два существенных преимущества. Во-первых, вы получаете возможность исправить ошибку. Во-вторых, предотвращается автоматическое прерывание выполнения программы. Большинство пользователей будут недовольны (и это как минимум), если ваша программа будет останавливаться и выводить трассировку стека всякий раз при возникновении ошибки. К счастью, предотвратить это достаточно просто.

Чтобы противостоять этому и обрабатывать ошибки времени выполнения, нужно просто поместить код, который вы хотите наблюдать, внутрь блока `try`. Непосредственно за блоком `try` следует включить конструкцию `catch`, которая задает тип перехватываемого исключения. Чтобы проиллюстрировать, насколько это просто делается, в следующую программу включен блок `try` с конструкцией `catch`, который обрабатывает исключение `ArithmeticException`, создаваемое в результате попытки деления на нуль.

```
class Exc2 {
    public static void main(String args[]) {
        int d, a;

        try { // Мониторинг блока кода.
            d = 0;
            a = 42 / d;
            System.out.println("Это не будет выведено.");
        } catch (ArithmeticException e) { // перехват ошибки
                                          // деления на нуль
            System.out.println("Деление на нуль.");
        }
        System.out.println("После оператора catch.");
    }
}
```

Эта программа создает следующий вывод.

```
Деление на нуль.
После оператора catch.
```

Обратите внимание на то, что вызов метода `println()` внутри блока `try` никогда не будет выполняться. Как только исключение передано, управление передается из блока `try` в блок `catch`. То есть строка "Это не будет выведено" не отображается. После того как блок `catch` будет выполнен, управление передается на строку программы, следующую за всем блоком `try/catch`.

Операторы `try` и `catch` составляют единый узел. Область действия блока `catch` не распространяется на те операторы, которые идут перед оператором `try`. Оператор `catch` не может перехватить исключение, переданное другим оператором `try` (кроме случаев вложенных конструкций `try`, которые будут описа-

ны ниже). Операторы, которые защищены блоком try, должны быть заключены в фигурные скобки (т.е. они должны находиться внутри блока). Вы не можете применить оператор try к отдельному оператору программы.

Целью правильно построенных операторов catch является разрешение исключительных ситуаций и продолжение работы, как если бы ошибки вообще не случались. Например, в следующей программе каждая итерация цикла for получает два случайных числа. Эти два числа делятся одно на другое, а результат используется для деления значения 12345. Окончательный результат помещается в переменную a. Если какая-либо из операций деления вызывает ошибку деления на нуль, эта ошибка перехватывается, значение переменной a устанавливается равным 0 и выполнение программы продолжается.

```
// Обработка исключения с продолжением работы.
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; // присвоить нуль и продолжить работу
            }
            System.out.println("a: " + a);
        }
    }
}
```

Отображение описания исключения

Класс Throwable переопределяет метод toString() (определенный в классе Object) таким образом, что он возвращает строку, содержащую описание исключения. Вы можете отобразить это описание с помощью метода println(), просто передав исключение в виде аргумента. Например, блок catch из предыдущего примера может быть переписан следующим образом.

```
catch (ArithmeticException e) {
    System.out.println("Исключение: " + e);
    a = 0; // присвоить нуль и продолжить работу
}
```

Когда эта версия подставляется в программу и программа запускается, каждая попытка деления на нуль отобразит следующее сообщение.

```
Исключение: java.lang.ArithmeticException: / by zero
```

Хотя в данном контексте это не имеет особого значения, все же возможность отобразить описание исключения в некоторых случаях полезна — в частности, когда вы экспериментируете с исключениями или занимаетесь отладкой.

Множественные операторы catch

В некоторых случаях один фрагмент кода может инициировать более одного исключения. Чтобы справиться с такой ситуацией, вы можете задать два или более операторов catch, каждый для перехвата своего типа исключений. Когда передается исключение, каждый оператор catch проверяется по порядку, и первый из них, тип которого соответствует исключению, выполняется. После того как выполнится один из операторов catch, все остальные пропускаются и выполнение программы продолжается с места, следующего за блоком try/catch. В следующем примере кода перехватываются два разных типа исключений.

```
// Демонстрация применения множественных операторов 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 0: " + e);
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Ошибка индекса массива: " + e);
        }
        System.out.println("После блока try/catch.");
    }
}
```

Эта программа вызовет исключение деления на нуль, если будет запущена без аргументов командной строки, поскольку в этом случае значение переменной a будет равно 0. Она выполнит деление, если будет передан аргумент командной строки, устанавливающий значение переменной a равным значению больше нуля. Но в этом случае будет создано исключение ArrayIndexOutOfBoundsException, так как длина массива целых чисел c равна 1, в то время как программа пытается присвоить значение элементу массива c[42].

Вот результаты запуска этой программы обоими способами.

```
C:\>java MultipleCatches
a = 0
Деление на 0: java.lang.ArithmeticException: / by zero
После блока try/catch.
C:\>java MultipleCatches TestArg
a = 1
Ошибка индекса массива: java.lang.ArrayIndexOutOfBoundsException:42
После блока try/catch.
```

Когда используются множественные операторы catch, важно помнить, что обработчики подклассов исключений должны следовать перед любыми обработчиками их суперклассов. Дело в том, что оператор catch, который использует суперкласс, будет перехватывать все исключения этого суперкласса плюс всех его подклассов. То есть исключения подкласса никогда не будут обработаны, если вы попытаетесь их перехватить после обработчика его суперкласса. Более того, в Java недостижимый код является ошибкой. Например, рассмотрим следующую программу.

```
/* Эта программа содержит ошибку.

   Подкласс должен идти перед его суперклассом в
   последовательности операторов catch. В противном случае
```

```
   будет создан недоступный код, что приведет к ошибке при компиляции.
*/
class SuperSubCatch {
    public static void main(String args[]) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch(Exception e) {
            System.out.println("Общий перехват Exception.");
        }
        /* Этот catch никогда не будет достигнут, потому что
           ArithmeticException - это подкласс Exception. */
        catch(ArithmeticException e) { // Ошибка - недостижимый код
            System.out.println("Это никогда не выполнится.");
        }
    }
}
```

Если вы попытаетесь скомпилировать эту программу, то получите сообщение об ошибке, говорящее о том, что второй оператор `catch` недостижим, потому что исключение уже перехвачено. Поскольку класс исключения `ArithmeticException` — подкласс класса `Exception`, первый оператор `catch` обработает все ошибки, основанные на классе `Exception`, включая `ArithmeticException`. Это означает, что второй оператор `catch` не будет никогда выполнен. Чтобы исправить это, потребуется изменить порядок следования операторов `catch`.

Вложенные операторы `try`

Операторы `try` могут быть вложенными. То есть оператор `try` может находиться внутри блока другого оператора `try`. Всякий раз, когда управление попадает в блок `try`, контекст этого исключения заталкивается в стек. Если вложенный оператор `try` не имеет обработчика `catch` для определенного исключения, стек "раскручивается" и проверяются на соответствие обработчики `catch` следующего (внешнего) блока `try`. Это продолжается до тех пор, пока не будет найден подходящий оператор `catch` либо пока не будут проверены все уровни вложенных операторов `try`. Если подходящий оператор `catch` не будет найден, то исключение обработает система времени выполнения Java. Ниже приведен пример, в котором используются вложенные операторы `try`.

```
// Пример вложенных операторов try.
class NestTry {
    public static void main(String args[]) {
        try {
            int a = args.length;

            /* Если не указаны параметры командной строки,
               следующий оператор создаст
               исключение деления на нуль. */
            int b = 42 / a;

            System.out.println("a = " + a);

            try { // вложенный блок try
                /* Если используется один аргумент командной строки,
                   то исключение деления на нуль
                   будет создано следующим кодом. */
                if(a==1) a = a/(a-a); // деление на нуль
```

```
                /* Если используется два аргумента командной строки,
                   то создается исключение выхода за пределы массива.
                */
                if(a==2) {
                    int c[] = { 1 };
                    c[42] = 99; // создается исключение выхода
                                // за пределы массива
                }
            } catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Индекс за пределами массива: " + e);
            }

        } catch(ArithmeticException e) {
            System.out.println("Деление на 0: " + e);
        }
    }
}
```

Как видите, в этой программе один блок `try` вложен в другой. Программа работает следующим образом. Когда вы запускаете ее без аргументов командной строки, внешним блоком `try` создается исключение деления на нуль. Запуск программы с одним аргументом приводит к передаче исключения деления на нуль во вложенном блоке `try`.

Поскольку вложенный блок не обрабатывает это исключение, оно передается внешнему блоку `try`, который обрабатывает его. Если программе передается два аргумента командной строки, то создается исключение выхода индекса за границы массива во внутреннем блоке `try`. Вот примеры запуска этой программы, иллюстрирующие каждый случай.

```
C:\>java NestTry
Деление на 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Деление на 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
Индекс за пределами массива:
java.lang.ArrayIndexOutOfBoundsException:42
```

Вложение операторов `try` может быть не столь очевидным, если в процессе выполняются вызовы методов. Например, вы можете в пределах блока `try` вызывать метод, а внутри этого метода иметь еще один блок `try`. В этом случае блок `try` в теле метода находится внутри внешнего блока `try`, который вызывает этот метод. Ниже представлена версия предыдущей программы с блоком `try`, перемещенным внутрь метода `nesttry()`.

```
/* Операторы try могут быть неявно вложены в вызовах методов. */
class MethNestTry {
    static void nesttry(int a) {
        try { // вложенный блок try
            /* Если используется один аргумент командной строки,
               то исключение деления на нуль
                будет создано следующим кодом. */
            if(a==1) a = a/(a-a); // деление на нуль

                /* Если используется два аргумента командной строки,
                   то создается исключение выхода за пределы массива.
                */
            if(a==2) {
                int c[] = { 1 };
```

```
            c[42] = 99; // создается исключение выхода за
                        // пределы массива
        }
    } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Индекс за пределами массива: " + e);
    }
}

public static void main(String args[]) {
    try {
        int a = args.length;

        /* Если не указаны параметры командной строки,
           следующий оператор создаст
           исключение деления на нуль. */
        int b = 42 / a;
        System.out.println("a = " + a);
        nesttry(a);
    } catch(ArithmeticException e) {
        System.out.println("Деление на 0: " + e);
    }
}
}
```

Вывод этой программы идентичен предыдущему примеру.

Оператор `throw`

До сих пор мы перехватывали только те исключения, которые передавала система времени выполнения Java. Однако существует возможность передавать исключения из ваших программ явным образом, используя оператор `throw`. Его общая форма показана ниже.

```
throw экземпляр_Throwable;
```

Здесь *экземпляр_Throwable* должен быть объектом класса `Throwable` либо подклассом класса `Throwable`. Элементарные типы, такие как `int` или `char`, как и классы, отличные от класса `Throwable`, например классы `String` и `Object`, не могут быть использованы для исключений.

Существует два способа получить объект класса `Throwable`: с использованием параметра в операторе `catch` либо за счет создания объекта оператором `new`.

Поток выполнения останавливается непосредственно после оператора `throw` — любые последующие операторы не выполняются. Обнаруживается ближайший закрытый блок `try`, имеющий оператор `catch` соответствующего исключению типа. Если соответствие найдено, управление передается этому оператору. Если же нет, проверяется следующий внешний блок `try` и т.д. Если не находится подходящего по типу оператора `catch`, то стандартный обработчик исключений прерывает программу и выводит трассировку стека.

Ниже приведен пример программы, создающей и передающей исключение. Обработчик, который перехватывает его, повторно передает его для внешнего обработчика.

```
// Демонстрация применения throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Перехвачено внутри demoproc.");
```

```
            throw e; // повторно передать исключение
        }
    }
        public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Повторный перехват: " + e);
        }
    }
}
```

Эта программа получает две возможности обработки одной и той же ошибки. Сначала метод `main()` устанавливает контекст исключения, затем вызывает метод `demoproc()`, который устанавливает другой контекст обработки исключения и немедленно передает новый экземпляр исключения `NullPointerException`, который перехватывается в следующей строке. Затем исключение передается повторно. Ниже показан результирующий вывод.

```
Перехвачено внутри demoproc.
Повторный перехват: java.lang.NullPointerException: demo
```

Эта программа также демонстрирует, как создавать собственные объекты стандартных исключений Java. Обратите внимание на следующую строку.

```
throw new NullPointerException("demo");
```

Здесь оператор `new` используется для создания экземпляра исключения `NullPointerException`. Многие из встроенных исключений времени выполнения Java имеют, по меньшей мере, два конструктора: без параметров и со строковым параметром. Когда применяется вторая форма, аргумент указывает строку, описывающую исключение. Эта строка отображается, когда объект используется в качестве аргумента методов `print()` или `println()`. Она также может быть получена вызовом метода `getMessage()`, который определен в классе `Throwable`.

Оператор `throws`

Если метод может породить исключение, которое он сам не обрабатывает, он должен задать это поведение так, чтобы вызывающий его код мог позаботиться об этом исключении. Для этого к объявлению метода добавляется конструкция `throws`. Конструкция `throws` перечисляет типы исключений, которые метод может передавать. Это необходимо для всех исключений, кроме имеющих тип `Error`, `RuntimeException` либо их подклассов. Все остальные исключения, которые может передавать метод, должны быть объявлены в конструкции `throws`. Если этого не сделать, получится ошибка во время компиляции.

Вот общая форма объявления метода, которая включает оператор `throws`.

```
тип имя_метода(список_параметров) throws список_исключений
{
    // тело метода
}
```

Здесь *список_исключений* — это разделенный запятыми список исключений, которые метод может передать.

Ниже представлен пример неправильной программы, пытающейся передать исключение, которое сама она не перехватывает. Поскольку в программе не указан оператор `throws` для отображения этого факта, такая программа не компилируется.

```
// Эта программа содержит ошибку и потому не компилируется.
class ThrowsDemo {
```

```
    static void throwOne() {
        System.out.println("Внутри throwOne.");
        throw new IllegalAccessException("демо");
    }
    public static void main(String args[]) {
        throwOne();
    }
}
```

Чтобы откомпилировать этот пример, нужно внести в него два изменения. Во-первых, следует объявить, что метод `throwOne()` передает исключение `IllegalAccessException`. Во-вторых, метод `main()` должен определять блок `try/catch`, который перехватит это исключение.

Исправленный пример выглядит следующим образом.

```
// Теперь код корректен.
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Внутри throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Перехвачено " + e);
        }
    }
}
```

Вот результат, полученный при запуске этой программы.

```
Внутри throwOne
Перехвачено java.lang.IllegalAccessException: demo
```

Оператор `finally`

Когда исключение передано, выполнение метода направляется по нелинейному пути, изменяющему нормальный поток управления внутри метода. В зависимости от того, как написан метод, существует даже возможность преждевременного возврата управления. В некоторых методах это может служить причиной серьезных проблем. Например, если метод при входе открывает файл и закрывает его при выходе, вероятно, вы не захотите, чтобы выполнение кода, закрывающего файл, было пропущено из-за применения механизма обработки исключений. Ключевое слово `finally` предназначено для того, чтобы справиться с такой ситуацией.

Ключевое слово `finally` создает блок кода, который будет выполнен после завершения блока `try/catch`, но перед кодом, следующим за ним. Блок `finally` выполняется независимо от того, передано исключение или нет. Если исключение передано, блок `finally` выполняется, даже если ни один оператор `catch` этому исключению не соответствует. В любой момент, когда метод собирается возвратить управление вызывающему коду изнутри блока `try/catch` (из-за необработанного исключения или явным применением оператора `return`), блок `finally` будет выполнен перед возвратом управления из метода. Это может быть удобно для закрытия файловых дескрипторов либо освобождения других ресурсов, которые были получены в начале метода и должны быть освобождены перед возвратом. Оператор `finally` необязателен. Однако каждый оператор `try` требует наличия, по крайней мере, одного оператора `catch` или `finally`. Ниже приведен пример программы, которая показывает три метода, возвращающих управление разными способами, но ни один из них не пропускает выполнения блока `finally`.

```
class FinallyDemo {
    // Передает исключение из метода.
    static void procA() {
        try {
            System.out.println("внутри procA");
            throw new RuntimeException("демо");
        } finally {
            System.out.println("блок finally procA");
        }
    }

    // Возврат управления в блоке try.
    static void procB() {
        try {
            System.out.println("внутри procB");
            return;
        } finally {
            System.out.println("блок finally procB");
        }
    }

    // Нормальное выполнение блока try.
    static void procC() {
        try {
            System.out.println("внутри procC");
        } finally {
            System.out.println("блок finally procC");
        }
    }

    public static void main(String args[]) {
        try {
            procA();
        } catch (Exception e) {
            System.out.println("Исключение перехвачено");
        }

        procB();
        procC();
    }
}
```

В этом примере метод `procA()` преждевременно прерывает выполнение в блоке `try`, передавая исключение. Блок `finally` все равно выполняется. В методе `procB()` возврат управления осуществляется в блоке `try` оператором `return`. Блок `finally` выполняется перед возвратом из метода `procB()`. В методе `procC()` блок `try` выполняется нормально, без ошибок. Однако блок `finally` выполняется все равно.

Помните! Если блок `finally` ассоциируется с блоком `try`, то блок `finally` будет выполнен по завершении блока `try`.

Вот результат, созданный предыдущей программой.

```
внутри procA
блок finally procA
Исключение перехвачено
внутри procB
блок finally procB
внутри procC
блок finally procC
```

Встроенные исключения Java

Внутри стандартного пакета `java.lang` определено несколько классов исключений. Некоторые из них использовались в предыдущих примерах. Большинство из этих исключений являются подклассами стандартного типа `RuntimeException`. Как уже объяснялось ранее, эти исключения не нужно включать в список `throws` метода – они называются *непроверяемыми исключениями*, поскольку компилятор не проверяет факт обработки или передачи методом таких исключений. Непроверяемые исключения, определенные в пакете `java.lang`, описаны в табл. 10.1. В табл. 10.2 перечислены те определенные в пакете `java.lang` исключения, которые должны быть включены в списки `throws` методов, которые могут их создавать и не обрабатывают самостоятельно. Они называются *проверяемыми исключениями*. В Java также определено несколько других типов исключений, имеющих отношение к библиотекам классов.

Таблица 10.1. Непроверяемые подклассы класса `RunTimeException`, определенные в пакете `java.lang`

Исключение	Описание
`ClassNotFoundException`	Класс не найден
`CloneNotSupportedException`	Попытка клонировать объект, который не реализует интерфейс `Cloneable`
`IllegalAccessException`	Доступ к классу не разрешен
`InstantiationException`	Попытка создать объект абстрактного класса или интерфейса
`InterruptedException`	Один поток прерван другим потоком
`NoSuchFieldException`	Запрошенное поле не существует
`NoSuchMethodException`	Запрошенный метод не существует
`ReflectiveOperationException`	Суперкласс исключений, связанных с рефлексией. (Добавлено в JDK 7)

Таблица 10.2. Проверяемые исключения, определенные в пакете `java.lang`

Исключение	Описание
`ArithmeticException`	Арифметическая ошибка, такая как деление на нуль
`ArrayIndexOutOfBoundsException`	Выход индекса за границу массива
`ArrayStoreException`	Присваивание элементу массива объекта несовместимого типа
`ClassCastException`	Неверное приведение
`EnumConstantNotPresentException`	Попытка использования неопределенного значения перечисления
`IllegalArgumentException`	Неверный аргумент использован при вызове метода
`IllegalMonitorStateException`	Неверная операция мониторинга, такая как ожидание незаблокированного потока
`IllegalStateException`	Среда или приложение в некорректном состоянии

Окончание табл. 10.2

Исключение	Описание
`IllegalThreadStateException`	Запрошенная операция несовместима с текущим состоянием потока
`IndexOutOfBoundsException`	Некоторый тип индекса вышел за допустимые пределы
`NegativeArraySizeException`	Создан массив отрицательного размера
`NullPointerException`	Неверное использование пустой ссылки
`NumberFormatException`	Неверное преобразование строки в числовой формат
`SecurityException`	Попытка нарушения безопасности
`StringIndexOutOfBounds`	Попытка использования индекса за пределами строки
`TypeNotPresentException`	Тип не найден
`UnsupportedOperationException`	Обнаружена неподдерживаемая операция

Создание собственных подклассов исключений

Хотя встроенные исключения Java обрабатывают большинство распространенных ошибок, вероятно, вам потребуется создать ваши собственные типы исключений для обработки ситуаций, специфичных для ваших приложений. Это достаточно просто сделать: определите подкласс класса `Exception` (который, разумеется, является подклассом класса `Throwable`). Ваши подклассы не обязаны реализовать что-либо — важно само их присутствие в системе типов, что позволит использовать их как исключения.

Класс `Exception` не определяет никаких собственных методов. Естественно, он наследует методы, представленные в классе `Throwable`. Таким образом, всем исключениям, включая те, что вы создадите сами, доступны методы, определенные в классе `Throwable`. Все они перечислены в табл. 10.3. Вы можете также переопределить один или несколько этих методов в собственных классах исключений.

Таблица 10.3. Методы, определенные в классе `Throwable`

Метод	Описание
`final void addSuppressed(Throwable` *исключение*`)`	Добавляет *исключение* в список подавляемых исключений, связанный с вызывающим исключением. Используется, прежде всего, с новым оператором `try`-*с-ресурсами*. (Добавлено в JDK 7)
`Throwable fillInStackTrace()`	Возвращает объект класса `Throwable`, содержащий полную трассировку стека. Этот объект может быть передан повторно
`Throwable getCause()`	Возвращает исключение, лежащее под текущим исключением. Если такого нет, возвращается значение `null`
`String getLocalizedMessage()`	Возвращает локализованное описание исключения
`String getMessage()`	Возвращает описание исключения

Окончание табл. 10.3

Метод	Описание
`StackTraceElement[] getStackTrace()`	Возвращает массив, содержащий трассировку стека и состоящий из элементов класса `StackTraceElement`. Метод в верхушке стека – это метод, который был вызван непосредственно перед тем, как было передано исключение. Этот метод содержится в первом элементе массива. Класс `StackTraceElement` дает вашей программе доступ к информации о каждом элементе в трассировке, такой как имя его метода
`final Throwable[] getSuppressed()`	Получает подавленные исключения, связанные с вызывающим исключением, и возвращает массив, который содержит результат. Подавленные исключения создаются, прежде всего, новым оператором *try-с-ресурсами*. (Добавлено в JDK 7)
`Throwable initCause(Throwable` *исключение*`)`	Ассоциирует исключение с вызывающим исключением, как причиной этого вызывающего исключения. Возвращает ссылку на исключение
`void printStackTrace()`	Отображает трассировку стека
`void printStackTrace(PrintStream` *поток*`)`	Посылает трассировку стека в заданный поток
`void printStackTrace(PrintWriter` *поток*`)`	Посылает трассировку стека в заданный поток
`void setStackTrace(StackTraceElement` *элементы*`[])`	Устанавливает трассировку стека для *элементов*. Этот метод предназначен для специализированных приложений, а не для нормального применения
`String toString()`	Возвращает объект класса `String`, содержащий описание исключения. Этот метод вызывается из метода `println()` при выводе объекта класса `Throwable`

В следующем примере объявляется новый подкласс класса `Exception`, который затем используется для сообщения об ошибке в методе. Он переопределяет метод `toString()`, позволяя отобразить тщательно настроенное описание исключения.

```
// Эта программа создает пользовательский 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("Вызван 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 " + e);
        }
    }
}
```

В этом примере определен подкласс `MyException` класса `Exception`. Этот подкласс достаточно прост: он имеет только конструктор и переопределенный метод `toString()`, отображающий значение исключения. Класс `ExceptionDemo` определяет метод `compute()`, который передает объект исключения `MyException`. Это исключение передается, когда целочисленный параметр метода `compute()` принимает значение больше 10.

Метод `main()` устанавливает обработчик исключений `MyException`, затем вызывает метод `compute()` с правильным параметром (меньше 10) и неправильным, чтобы продемонстрировать оба пути выполнения кода. Ниже показан результат.

```
Вызван compute(1)
Нормальное завершение
Вызван compute(20)
Перехвачено MyException[20]
```

Сцепленные исключения

Начиная с J2SE 1.4 в подсистему исключений было добавлено такое средство, как *сцепленное исключение* (chained exception). Это средство позволяет ассоциировать с одним исключением другое, которое описывает причину появления первого. Например, представьте ситуацию, когда метод передает исключение `ArithmeticException`, поскольку была предпринята попытка деления на нуль. Однако реальная причина проблемы заключается в ошибке ввода-вывода, что приводит к неправильному делению. И хотя метод должен передать исключение `ArithmeticException`, так как произошла именно эта ошибка, вы можете также позволить вызывающему коду узнать о том, что в основе лежит ошибка ввода-вывода. Сцепленные исключения позволяют справиться с этой, а также с любой другой ситуацией, в которой присутствуют уровни исключений.

Чтобы разрешить сцепленные исключения, в класс `Throwable` были добавлены два конструктора и два метода.

Ниже показаны конструкторы.

```
Throwable(Throwable причинаИскл)
Throwable(String сообщение, Throwable причинаИскл)
```

В первой форме *причинаИскл* — это исключение, послужившее причиной текущего исключения. Таким образом, *причинаИскл* — это основная причина исключения. Вторая форма позволяет задать описание, а также определить причину исключения. Эти два конструктора были также добавлены в классы `Error`, `Exception` и `RuntimeException`.

Для сцепления исключений в класс `Throwable` были добавлены методы `getCause()` и `initCause()`. Они представлены в табл. 10.3 и повторяются здесь при обсуждении.

```
Throwable getCause()
Throwable initCause(Throwable причинаИскл)
```

Метод `getCause()` возвращает исключение, являющееся причиной текущего исключения. Если такого исключения нет, возвращается значение `null`. Метод `initCause()` ассоциирует исключение *причинаИскл* с вызывающим исключением и возвращает ссылку на исключение. Таким образом, вы можете ассоциировать причину с исключением уже после того, как исключение было создано. Однако причина исключения может быть задана только однажды. Таким образом, вы можете вызвать метод `initCause()` только однажды для каждого объекта исключения. Кроме того, если причина исключения была установлена конструктором, то вы не можете установить ее снова, используя метод `initCause()`. Вообще, метод `initCause()` используется для установки причин устаревших классов исключений, которые не поддерживают два описанных ранее дополнительных конструктора.

Вот пример, демонстрирующий применение механизма сцепления исключений.

```
// Демонстрация 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("Исходная причина: " + e.getCause());
        }
    }
```

Эта программа создает следующий вывод.

```
Перехвачено: java.lang.NullPointerException: верхний уровень
Исходная причина: java.lang.ArithmeticException: причина
```

В этом примере исключением верхнего уровня является `NullPointerException`. К нему добавлено исключение-причина — `ArithmeticException`. Когда исключение передается из метода `demoproc()`, оно перехватывается в методе `main()`. Затем исключение верхнего уровня отображается, а за ним следует лежащее в основе исключение, которое извлекается методом `getCause()`.

Сцепленные исключения могут вкладываться на любую глубину. То есть причина исключения может иметь собственную причину. Но имейте в виду, что слишком

длинные цепочки сцепленных исключений, скорее всего, свидетельствуют о плохом дизайне.

Сцепленные исключения не являются тем, что совершенно необходимо в каждой программе. Однако в случаях, когда информация об исключении-причине все-таки нужна, они представляют собой элегантное решение.

Три новых средства исключений JDK 7

В систему исключений комплекта JDK 7 добавлено три интересных и полезных средства. Первое автоматизирует процесс освобождения ресурса, такого как файл, когда он больше не нужен. Оно основано на расширенной форме оператора `try`, называемой оператором *try-с-ресурсами*, и описывается в главе 13 при рассмотрении файлов. Второе новое средство называется *мультиобработчик* (multi-catch), а третье иногда упоминается как *финальная повторная передача* (final rethrow) или *более точная повторная передача* (more precise rethrow). Последние два средства описаны здесь.

Мультиобработчик позволяет обработать несколько исключений в том же операторе `catch`. Вполне обычна ситуация, когда обработчики нескольких исключений используют одинаковый код, хотя они соответствуют разным исключениям. Теперь, вместо индивидуальной обработки исключения каждого типа, вы можете использовать один блок `catch` для обработки всех исключений без дублирования кода.

Чтобы использовать мультиобработчик, отделите в операторе `catch` каждый тип исключения оператором OR. Каждый параметр мультиобработчика неявно финальный. (При желании вы можете явно указать ключевое слово `final`, но необходимости в этом нет.) Поскольку каждый параметр мультиобработчика неявно финальный, ему не может быть присвоено новое значение.

Вот оператор `catch`, использующий мультиобработчик для обработки исключений `ArithmeticException` и `ArrayIndexOutOfBoundsException`.

```
catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
```

Следующая программа демонстрирует мультиобработчик в действии.

```
// Демонстрация мультиобработчика JDK 7.
class MultiCatch {
    public static void main(String args[]) {
        int a=10, b=0;
        int vals[] = { 1, 2, 3 };

        try {
            int result = a / b; // создает ArithmeticException
         // vals[10] = 19;      // создает ArrayIndexOutOfBoundsException
            // Этот оператор catch обработает оба исключения.
        } catch(ArithmeticException | ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Обрабатывается исключение: " + e);
        }
        System.out.println("После мультиобработчика.");
    }
}
```

Программа создаст исключение `ArithmeticException` при попытке деления на нуль. Если вы закомментируете оператор деления и снимете комментарий со следующей строки, будет создано исключение `ArrayIndexOutOfBoundsException`. Оба исключения обрабатываются одним оператором `catch`.

Средство более точной повторной передачи ограничивает тип исключений, которые могут быть повторно переданы только теми проверяемыми исключениями,

которые связаны с блоком передачи try, не обрабатываются приведенным выше оператором catch и являются подтипом или супертипом параметра. Хотя необходимость в этом средстве, возможно, возникнет и не часто, теперь оно доступно для использования. При применении более точной повторной передачи, параметр оператора catch должен фактически быть финальным, это значит, что ему либо не должно присваиваться новое значение в блоке catch, либо он должен быть явно объявлен как final.

Использование исключений

Обработка исключений представляет собой мощный механизм для управления сложными программами, которые имеют много динамических характеристик времени выполнения. Средства try, throws и catch следует считать простым способом обработки ошибок и необычных критических условий в вашей программной логике. В отличие от ряда других языков, в которых для индикации сбоев используются коды ошибок, в языке Java применяются исключения. Иными словами, когда метод может завершиться сбоем, он передает исключение. Это более простой способ справиться с ошибочными ситуациями.

Последнее замечание: операторы управления исключениями Java не должны рассматриваться как общий способ нелокального ветвления. Если вы будете это делать, то только запутаете ваш код и усложните его сопровождение.

ГЛАВА 11

Многопоточное программирование

В отличие от некоторых языков программирования, Java предлагает встроенную поддержку *многопоточного программирования*. Многопоточная программа содержит две или более частей, которые могут выполняться одновременно. Каждая часть такой программы называется *потоком* (thread), и каждый поток задает отдельный путь выполнения. Другими словами, многопоточность – это специализированная форма многозадачности.

Вы наверняка знакомы с многозадачностью, поскольку она поддерживается практически всеми современными операционными системами. Однако существует два отдельных типа многозадачности: многозадачность, основанная на процессах, и многозадачность, основанная на потоках. Важно понимать разницу между ними. Большинству читателей более знакома многозадачность, основанная на процессах. *Процесс* по сути своей – это выполняющаяся программа. То есть *многозадачность, основанная на процессах*, представляет собой средство, которое позволяет вашему компьютеру одновременно выполнять две или более программ. Так, например, основанная на процессах многозадачность позволяет запускать компилятор Java в то самое время, когда вы используете текстовый редактор или посещаете веб-сайт. В многозадачности, основанной на процессах, программа представляет собой наименьший элемент кода, которым может управлять планировщик операционной системы.

В среде *поточной многозадачности* наименьшим элементом управляемого кода является поток. Это означает, что одна программа может выполнять две или более задач одновременно. Например, текстовый редактор может форматировать текст в то же время, когда выполняется его печать, – до тех пор, пока эти два действия выполняются двумя отдельными потоками. То есть многозадачность на основе процессов имеет дело с "картиной в целом", а потоковая многозадачность справляется с деталями.

Многозадачные потоки требуют меньше накладных расходов, чем многозадачные процессы. Процессы – это тяжеловесные задачи, каждая из которых требует собственного адресного пространства. Межпроцессные коммуникации дорогостоящи и ограничены. Переключение контекста от одного процесса к другому также обходится дорого. С другой стороны, потоки проще. Они совместно используют одно и то же адресное пространство и один и тот же тяжеловесный процесс. Коммуникации между потоками экономны, а переключения контекста между потоками характеризуются низкой стоимостью. Хотя программы Java используются в многозадачных средах на основании процессов, такая многозадачность средствами Java не контролируется. А вот многопоточная многозадачность средствами Java контролируется.

Многопоточность позволяет вам писать эффективные программы, которые по максимуму используют доступную мощь процессора системы. Еще одним преимуществом многопоточности является сведение к минимуму времени ожидания. Это особенно важно для интерактивных сетевых сред, в которых работает Java, так как в них наличие ожидания и простоев – обычное явление. Например, скорость переда-

чи данных по сети намного ниже, чем скорость, с которой компьютер может их обрабатывать. Даже чтение и запись ресурсов локальной файловой системы намного медленнее, чем темп их обработки в процессоре. И, конечно, пользователь намного медленнее вводит данные с клавиатуры, чем их может обработать компьютер. В однопоточных средах ваша программа вынуждена ожидать окончания таких задач, прежде чем переходить к следующей, — даже если бо́льшую часть времени программа простаивает, ожидая ввода. Многопоточность помогает сократить время простоя, поскольку другие потоки могут выполняться, пока один ожидает.

Если вы программировали для таких операционных систем, как Windows, это значит, что вы уже знакомы с многопоточным программированием. Однако тот факт, что Java управляет потоками, делает многопоточность особенно удобной, поскольку многие детали подконтрольны вам как программисту.

Модель потоков Java

Система времени выполнения Java зависит от потоков во многих отношениях, и все библиотеки классов спроектированы с учетом многопоточности. Фактически Java использует потоки для того, чтобы обеспечить асинхронность всей среде выполнения. Это позволяет снизить неэффективность за счет предотвращения бесполезной растраты циклов центрального процессора.

Значение многопоточной среды лучше понимается при сравнении. Однопоточные системы используют подход, называемый *циклом событий с опросом*. В этой модели единственный поток управления выполняется в бесконечном цикле, опрашивая единственную очередь событий, чтобы принять решение о том, что делать дальше. Как только этот механизм опроса возвращает, скажем, сигнал о том, что сетевой файл готов к чтению, цикл событий передает управление соответствующему обработчику событий. До тех пор, пока тот не вернет управление, в программе ничего не может произойти. Это расходует время процессора. Это также может привести к тому, что одна часть программы будет доминировать над другими и не позволять обрабатывать любые другие события. Вообще говоря, в однопоточном окружении, когда поток блокируется (то есть приостанавливает выполнение) по причине ожидания некоторого ресурса, выполнение всей программы приостанавливается.

Выгода от многопоточности состоит в том, что основной механизм циклического опроса исключается. Один поток может быть приостановлен без остановки других частей программы. Например, время ожидания при чтении данных из сети либо ожидание пользовательского ввода может быть утилизировано где угодно. Многопоточность позволяет циклам анимации "засыпать" на секунду между показом соседних кадров, не приостанавливая работы всей системы. Когда поток блокируется в программе Java, то останавливается только один-единственный заблокированный поток. Все остальные потоки продолжают выполняться.

За последние несколько лет многоядерные системы стали вполне обычным явлением. Конечно, одноядерные системы все еще широко распространены и используются. Важно понимать, что многопоточные средства Java работают в обоих типах систем. В одноядерной системе одновременно выполняющиеся потоки совместно используют процессор, получая для каждого потока некий сектор процессорного времени. Поэтому в одноядерной системе два или более потоков фактически не выполняются одновременно, они ожидают своей очереди на использование процессорного времени. Но в многоядерных системах два или более потоков фактически могут выполняться одновременно. Во многих случаях это может увеличить эффективность программы и повысить скорость определенных операций.

На заметку! В JDK 7 добавлена инфраструктура Fork/Join Framework — мощное средство создания многопоточных приложений с автоматическим масштабированием для лучшего использования многоядерных систем. Инфраструктура Fork/Join Framework — это часть поддержки Java для параллельного программирования, которое предоставляет технологии оптимизации некоторых типов алгоритмов параллельного выполнения в системах с несколькими процессорами. Более подробная информация об Fork/Join Framework и других утилитах параллельности приведена в главе 27, а здесь рассматриваются традиционные многопоточные возможности Java.

Потоки существуют в нескольких состояниях. Вот их общее описание. Поток может *выполняться*. Он может быть *готов к выполнению*, как только получит время центрального процессора. Работающий поток может быть *приостановлен*, что временно прекращает его активность. Выполнение приостановленного потока может быть *возобновлено*, позволяя ему продолжить работу с того места, где он был приостановлен. Поток может быть *заблокирован*, когда ожидает какого-то ресурса. В любой момент поток может быть *прерван*, что немедленно останавливает его выполнение. Однажды прерванный поток уже не может быть возобновлен.

Приоритеты потоков

Java присваивает каждому потоку приоритет, который определяет поведение данного потока по отношению к другим. Приоритеты потоков задаются целыми числами, определяющими относительный приоритет одного потока по сравнению с другими. Значение приоритета само по себе никакого смысла не имеет — более высокоприоритетный поток не выполняется быстрее, чем низкоприоритетный, когда он является единственным выполняемым потоком в данный момент. Вместо этого приоритет потока используется для принятия решения при переключении от одного выполняющегося потока к другому. Это называется *переключением контекста*. Правила, которые определяют, когда должно происходить переключение контекста, достаточно просты.

- *Поток может добровольно уступить управление.* Для этого можно явно уступить очередь выполнения, приостановить поток или блокировать на время ожидания ввода-вывода. При таком сценарии все прочие потоки проверяются, и ресурсы процессора передаются потоку с максимальным приоритетом, который готов к выполнению.
- *Поток может быть прерван другим, более приоритетным потоком.* В этом случае низкоприоритетный поток, который не занимает процессор, просто приостанавливается высокоприоритетным потоком, независимо от того, что он делает. В основном, высокоприоритетный поток выполняется, как только он этого "захочет". Это называется *вытесняющей многозадачностью* (или *многозадачностью с приоритетами*).

В случае, когда два потока, имеющие одинаковый приоритет, претендуют на цикл процессора, ситуация усложняется. Для таких операционных систем, как Windows, потоки с одинаковым приоритетом разделяют время в циклическом режиме. Для операционных систем других типов потоки с одинаковым приоритетом должны принудительно передавать управление своим "родственникам". Если они этого не делают, другие потоки не запускаются.

Внимание! Из-за разницы в способах переключения операционными системами потоковых контекстов могут возникать проблемы переносимости.

Синхронизация

Поскольку многопоточность дает вашим программам возможность асинхронного поведения, должен существовать способ обеспечить синхронизацию, когда в этом возникает необходимость. Например, если вы хотите, чтобы два потока взаимодействовали и совместно использовали сложную структуру данных, такую как связный список, то вы нуждаетесь в способе предотвращения конфликтов между ними. То есть следует предотвратить запись данных в одном потоке, когда другой занимается их чтением. Для этой цели в Java реализован элегантный трюк из старой модели межпроцессной синхронизации, а именно – *монитор*. Монитор – это управляющий механизм, впервые реализованный Чарльзом Энтони Ричардом Хоаром. Вы можете воспринимать монитор как очень маленький ящик, который принимает только один поток в единицу времени. Как только поток вошел в монитор, все другие потоки должны ждать, пока тот не покинет его. Таким образом, монитор может быть использован для защиты совместно используемых ресурсов от одновременного использования более чем одним потоком.

Большинство многопоточных систем применяет мониторы как объекты, которые ваша программа может получить и которыми она может манипулировать. Java предлагает более чистое решение. Не существует отдельного класса монитора вроде "Monitor". Вместо этого каждый объект имеет собственный неявный монитор, вход в который осуществляется автоматически, когда вызывается синхронизированный метод объекта. Когда поток находится внутри синхронизированного метода, ни один другой поток не может вызвать никакого синхронизированного метода этого объекта. Это позволяет вам писать очень ясный и краткий многопоточный код, поскольку поддержка синхронизации встроена в язык.

Обмен сообщениями

После того как вы разделите программу на отдельные потоки, вам нужно определить, как они будут общаться друг с другом. При программировании на некоторых других языках для установки взаимодействия между потоками вы вынуждены зависеть от операционной системы. То есть, конечно же, появляются накладные расходы. В отличие от этих языков, Java предоставляет ясный и экономичный способ общения двух или более потоков между собой – за счет вызова предопределенных методов, которыми обладают объекты. Система сообщений Java позволяет потоку войти в синхронизированный метод объекта и ожидать, пока какой-то другой поток явно не уведомит его о прибытии.

Класс `Thread` и интерфейс `Runnable`

Многопоточная система Java встроена в класс `Thread`, его методы и дополняющий его интерфейс `Runnable`. Класс `Thread` инкапсулирует поток выполнения. Поскольку вы не можете напрямую обратиться к нематериальному состоянию работающего потока, имеете дело с его *заместителем* (proxy) – экземпляром класса `Thread`, который породил его. Чтобы создать новый поток, ваша программа должна либо расширить класс `Thread`, либо реализовать интерфейс `Runnable`.

Класс `Thread` определяет несколько методов, которые помогают управлять потоками. Некоторые из них, которые будут упомянуты в настоящей главе, перечислены в табл. 11.1.

Таблица 11.1. Методы управления потоками класса `Thread`

Метод	Назначение
`getName`	Получить имя потока
`getPriority`	Получить приоритет потока
`isAlive`	Определить, выполняется ли поток
`join`	Ожидать завершения потока
`run`	Входная точка потока
`sleep`	Приостановить выполнение потока на заданное время
`start`	Запустить поток вызовом его метода `run()`

До сих пор во всех примерах книги использовался единственный поток управления. Далее в этой главе объясняется, как применять класс `Thread` и интерфейс `Runnable` для создания потоков и управления ими, начиная с потока, который есть в каждой программе Java, — главного.

Главный поток

Когда программа Java стартует, немедленно начинает выполняться один поток. Обычно его называют *главным потоком* (main thread) программы, потому что это тот поток, который запускается вместе с вашей программой. Главный поток важен по двум причинам.

- Это поток, от которого порождаются все "дочерние" потоки.
- Часто он должен быть последним потоком, завершающим выполнение, так как предпринимает различные завершающие действия.

Несмотря на то что главный поток создается автоматически при запуске программы, им можно управлять через объект класса `Thread`. Для этого следует получить ссылку на него вызовом метода `currentThread()`, который является открытым статическим (`public static`) методом класса `Thread`. Его общая форма выглядит следующим образом.

```
static Thread currentThread()
```

Этот метод возвращает ссылку на поток, из которого он был вызван. Получив ссылку на главный поток, вы можете управлять им точно так же, как любым другим. Рассмотрим следующий пример.

```
// 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("Мой 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 этой программе ссылка на текущий поток (в данном случае — главный) получается вызовом метода `currentThread()`, и эта ссылка сохраняется в локальной переменной `t`. Далее программа отображает информацию о потоке. Программа вызывает метод `setName()` для изменения внутреннего имени потока. После этого информация о потоке отображается заново. Далее в цикле выводятся цифры в обратном порядке с задержкой на 1 секунду после каждой строки. Пауза организуется вызовом метода `sleep()`. Аргумент метода `sleep()` задает период задержки в миллисекундах. Обратите внимание на блок `try/catch` вокруг цикла. Метод `sleep()` в классе `Thread` может передать исключение `InterruptException`. Это может произойти, если некоторый другой поток захочет прервать выполнение этого спящего потока. Данный пример просто выводит сообщение, если поток прерывается. В реальных программах вы будете обрабатывать подобную ситуацию иначе. Ниже показан вывод, создаваемый этой программой.

```
Текущий поток: Thread[main,5,main]
После изменения имени: Thread[My Thread,5,main]
5
4
3
2
1
```

Обратите внимание на то, что вывод создается, когда переменная `t` используется в качестве аргумента для метода `println()`. Он отображает по порядку имя потока, его приоритет и имя его группы. По умолчанию имя главного потока — `main`. Его приоритет равен 5, что является значением по умолчанию, а `main` — также имя группы потоков, к которой относится данный. Группа потоков — это структура данных, которая управляет состоянием набора потоков в целом. После того как имя потока изменено, переменная `t` выводится вновь. На этот раз отображается новое имя потока.

Давайте поближе взглянем на определенные в классе `Thread` методы, которые используются в программе. Метод `sleep()` заставляет поток, из которого он был вызван, приостановить выполнение на указанное количество миллисекунд. Его общая форма выглядит так.

```
static void sleep(long миллисекунды) throws InterruptedException
```

Количество миллисекунд, на которое нужно приостановить выполнение, передается в параметре *миллисекунды*. Этот метод может передать исключение `InterruptedException`.

Метод `sleep()` имеет также вторую форму, показанную ниже, которая позволяет задать период в миллисекундах и наносекундах.

```
static void sleep(long миллисекунды, long наносекунды) throws
InterruptedException
```

Вторая форма может применяться только в средах, которые предусматривают задание временных периодов в наносекундах.

Как показано в предыдущей программе, вы можете установить имя потока, используя метод `setName()`. Получить имя потока можно вызовом метода `getName()` (эта процедура в программе не показана). Эти методы являются членами класса `Thread` и объявлены следующим образом.

```
final void setName(String имя_потока)
final String getName()
```

Здесь *имя_потока* указывает имя потока.

Создание потока

В наиболее общем смысле вы создаете поток, реализуя объект класса Thread. В Java для этого определены два способа.

- С помощью реализации интерфейса Runnable.
- С помощью расширения класса Thread.

В следующих разделах рассматриваются эти способы по очереди.

Реализация интерфейса Runnable

Самый простой способ создания потока — это объявление класса, реализующего интерфейс Runnable. Интерфейс Runnable абстрагирует единицу исполняемого кода. Вы можете создать поток из любого объекта, реализующего интерфейс Runnable. Чтобы реализовать интерфейс Runnable, класс должен объявить единственный метод run().

```
public void run()
```

Внутри метода run() вы определяете код, который, собственно, составляет новый поток. Важно понимать, что метод run() может вызывать другие методы, использовать другие классы, объявлять переменные — точно так же, как это делает главный поток. Единственным отличием является то, что метод run() устанавливает точку входа для другого, параллельного потока внутри вашей программы. Этот поток завершится, когда метод run() вернет управление.

После того как будет объявлен класс, реализующий интерфейс Runnable, вы создадите объект типа Thread из этого класса. В классе Thread определено несколько конструкторов. Тот, который должен использоваться в данном случае, выглядит следующим образом.

```
Thread(Runnable объект_потока, String имя_потока)
```

В этом конструкторе *объект_потока* — это экземпляр класса, реализующего интерфейс Runnable. Он определяет, где начнется выполнение потока. Имя нового потока передается в параметре *имя_потока*.

После того как новый поток будет создан, он не запускается до тех пор, пока вы не вызовете метод start(), объявленный в классе Thread. По сути, метод start() выполняет вызов метода run(). Метод start() показан ниже.

```
void start()
```

Рассмотрим пример, создающий новый поток и запускающий его выполнение.

```
// Создание второго потока.
class NewThread implements Runnable {
    Thread t;

    NewThread() {
        // Создать новый, второй поток
        t = new Thread(this, "Демонстрационный поток");
        System.out.println("Дочерний поток создан: " + t);
        t.start(); // Запустить поток
    }

    // Точка входа 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("Дочерний поток завершен");
    }
}

class ThreadDemo {
    public static void main(String args[]) {
        new NewThread(); // создать новый 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("Главный поток завершен.");
    }
}
```

Внутри конструктора `NewThread()` в следующем операторе создается новый объект класса `Thread`.

```
t = new Thread(this, "Демонстрационный поток");
```

Передача объекта `this` в первом аргументе означает, что вы хотите, чтобы новый поток вызвал метод `run()` объекта `this`. Далее вызывается метод `start()`, в результате чего запускается выполнение потока начиная с метода `run()`. Это запускает цикл `for` дочернего потока. После вызова метода `start()` конструктор `NewThread()` возвращает управление методу `main()`. Когда главный поток продолжает свою работу, он входит в свой цикл `for`. После этого оба потока выполняются параллельно, совместно используя ресурсы процессора в одноядерной системе, вплоть до завершения своих циклов. Вывод, создаваемый этой программой, показан ниже (ваш вывод может варьироваться, в зависимости от конкретной среды исполнения).

```
Дочерний поток: Thread[Демонстрационный поток,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
Дочерний поток завершен.
Главный поток: 2
Главный поток: 1
Главный поток завершен.
```

Как уже упоминалось ранее, в многопоточной программе главный поток зачастую должен завершать выполнение последним. Фактически, для некоторых старых виртуальных машин Java (JVM), если главный поток завершается до завершения до-

черних потоков, то исполняющая система Java может "зависнуть". Предыдущая программа гарантирует, что главный поток завершится последним, поскольку главный поток "спит" 1000 миллисекунд между итерациями цикла, а дочерний поток — только 500 миллисекунд. Это заставляет дочерний поток завершиться раньше главного. Но далее вы узнаете лучший способ ожидания завершения потоков.

Расширение класса `Thread`

Еще один способ создания потока — это объявить класс, расширяющий класс `Thread`, а затем создать экземпляр этого класса. Расширяющий класс обязан переопределить метод `run()`, который является точкой входа для нового потока. Он также должен вызвать метод `start()` для запуска выполнения нового потока. Ниже приведен пример предыдущей программы, переписанной с использованием расширения класса `Thread`.

```
// Создание второго потока расширением Thread
class NewThread extends Thread {

    NewThread() {
        // Создать новый второй поток
        super("Демонстрационный поток");
        System.out.println("Дочерний поток: " + this);
        start(); // Запустить поток
    }

    // Точка входа 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("Дочерний поток завершен.");
    }
}

class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // Создать новый 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("Главный поток завершен.");
    }
}
```

Эта программа создает точно такой же вывод, что и предыдущая версия. Как вы можете видеть, дочерний поток создается при конструировании объекта класса `NewThread`, который наследуется от класса `Thread`.

Обратите внимание на метод `super()` внутри класса `NewThread`. Он вызывает следующую форму конструктора `Thread()`.

```
public Thread(String имя_потока)
```

Здесь *имя_потока* указывает имя потока.

Выбор подхода

В данный момент вы можете спросить, почему Java предлагает два способа создания дочерних потоков и какой из них лучше. Ответы на эти вопросы взаимосвязаны. Класс `Thread` определяет несколько методов, которые могут быть переопределены в производных классах. Из этих методов только один *должен* быть переопределен в обязательном порядке — это метод `run()`. Конечно, этот же метод нужен, когда вы реализуете интерфейс `Runnable`. Многие программисты Java считают, что классы следует расширять только в случаях, когда они должны быть усовершенствованы или некоторым образом модифицированы. Поэтому если вы не переопределяете никаких других методов класса `Thread`, то вероятно, лучше просто реализовать интерфейс `Runnable`. Кроме того, при реализации интерфейса `Runnable` ваш класс потока не должен наследовать класс `Thread`, чтобы освободиться от наследования других классов. В конечном счете, какой из подходов использовать, остается на ваше усмотрение. Тем не менее далее в этой главе мы будем создавать потоки, используя классы, реализующие интерфейс `Runnable`.

Создание множества потоков

До сих пор вы использовали только два потока: главный и один дочерний. Однако ваша программа может порождать столько потоков, сколько необходимо. Например, в следующей программе создается 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);
        t.start(); // запустить поток
    }

    // Входная точка 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[]) {
        new NewThread("Один"); // запуск потоков
        new NewThread("Два");
        new NewThread("Три");

        try {
            // ожидание завершения других потоков
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван");
        }

        System.out.println("Главный поток завершен.");
    }
}
```

Пример вывода этой программы показан ниже (ваш вывод может отличаться, в зависимости от конкретной среды исполнения).

```
Новый поток: Thread[Один,5,main]
Новый поток: Thread[Два,5,main]
Новый поток: Thread[Три,5,main]
Один: 5
Два: 5
Три: 5
Один: 4
Два: 4
Три: 4
Один: 3
Три: 3
Два: 3
Один: 2
Три: 2
Два: 2
Один: 1
Три: 1
Два: 1
Один завершен.
Два завершен.
Три завершен.
Главный поток завершен.
```

Как видите, будучи запущенными, все три дочерних потока совместно используют ресурс центрального процессора. Обратите внимание на вызов метода `sleep(10000)` в методе `main()`. Это заставляет главный поток "уснуть" на 10 секунд и гарантирует, что он будет завершен последним.

Использование методов `isAlive()` и `join()`

Как упоминалось, зачастую необходимо, чтобы главный поток завершался последним. В предыдущих примерах это обеспечивается вызовом метода `sleep()` из метода `main()` с задержкой, достаточной для того, чтобы гарантировать, что все дочерние потоки завершатся раньше главного. Однако это неудовлетворительное решение, которое вызывает серьезный вопрос: как один поток может знать о том, что другой завершился? К счастью, класс `Thread` предлагает средство, которое дает ответ на этот вопрос.

Существует два способа определить, что поток был завершен. Во-первых, вы можете вызвать метод `isAlive()` для этого потока. Этот метод определен в классе `Thread`, и его общая форма такова.

```
final Boolean isAlive()
```

Метод `isAlive()` возвращает значение `true`, если поток, для которого он вызван, еще выполняется. В противном случае он возвращает значение `false`.

Во-вторых, существует метод, который вы будете использовать чаще, чтобы дождаться завершения потока, а именно — метод `join()`.

```
final void join() throws InetrruptedException
```

Этот метод ожидает завершения потока, для которого он вызван. Его имя отражает концепцию, что вызывающий поток ожидает, когда указанный поток присоединиться к нему. Дополнительные формы метода `join()` позволяют указывать максимальный период времени, который вы будете ожидать завершения указанного потока.

Ниже приведена усовершенствованная версия предыдущего примера, использующая метод `join()` для гарантии того, что главный поток завершился последним. Здесь также демонстрируется применение метода `isAlive()`.

```
// Применение join() для ожидания завершения 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);
        t.start(); // Запуск потока
    }

    // Входная точка 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 ob1 = new NewThread("Одни");
        NewThread ob2 = new NewThread("Два");
        NewThread ob3 = new NewThread("Три");

        System.out.println("Поток Один запущен: "
                           + ob1.t.isAlive());
        System.out.println("Поток Два запущен: "
                           + ob2.t.isAlive());
        System.out.println("Поток Три запущен: "
                           + ob3.t.isAlive());
        // ожидать завершения потоков
        try {
```

```
            System.out.println("Ожидание завершения потоков.");
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван");
        }

        System.out.println("Поток Один запущен: "
                           + ob1.t.isAlive());
        System.out.println("Поток Два запущен: "
                           + ob2.t.isAlive());
        System.out.println("Поток Три запущен: "
                           + ob3.t.isAlive());

        System.out.println("Главный поток завершен.");
    }
}
```

Пример вывода этой программы показан ниже (ваш вывод может отличаться, в зависимости от конкретной среды исполнения).

```
Новый поток: Thread[Одни,5,main]
Новый поток: Thread[Два,5,main]
Новый поток: Thread[Три,5,main]
Поток Один запущен: true
Поток Два запущен: true
Поток Три запущен: true
Ожидание завершения потоков.
Один: 5
Два: 5
Три: 5
Один: 4
Два: 4
Три: 4
Один: 3
Два: 3
Три: 3
Один: 2
Два: 2
Три: 2
Один: 1
Два: 1
Три: 1
Два завершен.
Три завершен.
Один завершен.
Поток Один запущен: false
Поток Два запущен: false
Поток Три запущен: false
Главный поток завершен.
```

Как видите, после того как вызовы метода `join()` вернут управление, потоки прекращают работу.

Приоритеты потоков

Планировщик потоков использует приоритеты потоков для принятия решения о том, когда каждому потоку будет разрешено работать. Теоретически высокоприоритетные потоки получают больше времени процессора, чем низкоприоритет-

ные. Практически объем процессорного времени, который получает поток, часто зависит от нескольких факторов, помимо его приоритета. (Например, то, как операционная система реализует многозадачность, может влиять на относительную доступность процессорного времени.) Высокоприоритетный поток может также выгружать низкоприоритетный. Например, когда низкоприоритетный поток работает, а высокоприоритетный собирается продолжить свою прерванную работу (в связи с приостановкой или ожиданием завершения операции ввода-вывода), то последний выгружает низкоприоритетный поток.

Теоретически потоки с одинаковым приоритетом должны получать равный доступ к центральному процессору. Но вы должны быть осторожны. Помните, что язык Java спроектирован для работы в широком спектре сред. Некоторые из этих сред реализуют многозадачность принципиально отлично от других. В целях безопасности потоки с одинаковым приоритетом должны получать управление в равной степени. Это гарантирует, что все потоки получат возможность выполняться в среде операционных систем с не вытесняющей многозадачностью. На практике, даже в средах с не вытесняющей многозадачностью, большинство потоков все-таки имеет шанс выполняться, поскольку неизбежно сталкиваются с блокирующими ситуациями, такими как ожидание ввода-вывода. Когда подобное случается, заблокированный поток приостанавливается и остальные потоки могут работать. Но если вы хотите добиться гладкой многопоточной работы, то не должны полагаться на это. К тому же некоторые типы задач интенсивно нагружают процессор. Такие потоки захватывают процессор. Потокам такого типа следует передавать управление от случая к случаю, чтобы позволить выполняться другим.

Чтобы установить приоритет потока, используйте метод `setPriority()` класса `Thread`. Так выглядит его общая форма.

```
final void setPriority(int уровень)
```

Здесь *уровень* задает новый уровень приоритета для вызывающего потока. Значение *уровень* должно быть в пределах диапазона от `MIN_PRIORITY` до `MAX_PRIORITY`. В настоящее время эти значения равны соответственно 1 и 10. Чтобы вернуть потоку приоритет по умолчанию, укажите `NORM_PRIORITY`, который в настоящее время равен 5. Эти приоритеты определены как статические финальные (`static final`) переменные в классе `Thread`.

Вы можете получить текущее значение приоритета потока, вызвав метод `getPriority()` класса `Thread`.

```
final int getPriority()
```

Реализации Java могут иметь принципиально разное поведение в том, что касается планирования потоков. Большинство несовпадений возникает, когда вы полагаетесь на вытесняющую многозадачность вместо совместного использования времени процессора. Наиболее безопасный способ получить предсказуемое межплатформенное поведение Java — это использовать потоки, которые принудительно осуществляют управление центральным процессором.

Синхронизация

Когда два или более потоков имеют доступ к одному совместно используемому ресурсу, они нуждаются в гарантии, что ресурс будет использован только одним потоком в одно и то же время. Процесс обеспечения этого называется *синхронизацией*. Как вы увидите, язык Java предлагает ее уникальную поддержку на уровне языка.

Ключом к синхронизации является концепция монитора. *Монитор* – это объект, который используется, как *взаимоисключающая блокировка* (mutually exclusive lock – mutex), или *мьютекс*. Только один поток может в одно и то же время *владеть* монитором. Когда поток запрашивает блокировку, говорят, что он *входит* в монитор. Все другие потоки, которые пытаются войти в заблокированный монитор, будут приостановлены до тех пор, пока первый поток не *выйдет* из монитора. Обо всех прочих потоках говорят, что они *ожидают* монитора. Поток, который владеет монитором, может повторно войти в него, если пожелает.

Вы можете синхронизировать ваш код двумя способами, которые предусматривают использование ключевого слова `synchronized`; здесь рассмотрим оба способа.

Использование синхронизированных методов

Синхронизация в Java проста, поскольку объекты имеют собственные, ассоциированные с ними неявные мониторы. Чтобы войти в монитор объекта, следует просто вызвать метод, модифицированный ключевым словом `synchronized`. Когда поток находится внутри синхронизированного метода, все другие потоки, которые пытаются вызвать его (или любые другие синхронизированные методы) в том же экземпляре, должны ожидать. Чтобы выйти из монитора и передать управление объектом другому ожидающему потоку, владелец монитора просто возвращает управление из синхронизированного метода.

Чтобы понять необходимость синхронизации, давайте начнем с простого примера, который не использует ее, хотя и должен. Следующая программа содержит три простых класса. Первый из них, `Callme`, имеет единственный метод – `call()`. Этот метод принимает параметр `msg` класса `String` и пытается вывести строку `msg` внутри квадратных скобок. Интересно отметить, что после того, как метод `call()` выводит открывающую скобку и строку `msg`, он вызывает метод `Thread.sleep(1000)`, который приостанавливает текущий поток на одну секунду.

Конструктор следующего класса, `Caller`, принимает ссылку на экземпляры классов `Callme` и `String`, которые сохраняются соответственно в переменных `target` и `msg`. Конструктор также создает новый поток, который вызовет метод `run()` объекта. Поток стартует немедленно. Метод `run()` класса `Caller` вызывает метод `call()` для экземпляра `target` класса `Callme`, передавая ему строку `msg`. Наконец, класс `Synch` начинает с создания единственного экземпляра класса `Callme` и трех экземпляров класса `Caller`, каждый с уникальной строкой сообщения. Один экземпляр класса `Callme` передается каждому конструктору `Caller()`.

```
// Эта программа не синхронизирована.
class Callme {
    void call(String msg) {
        System.out.print("[" + msg);
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {
            System.out.println("Прервано");
        }
        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);
        t.start();
    }
    public void run() {
        target.call(msg);
    }
}

class Synch {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Добро пожаловать");
        Caller ob2 = new Caller(target, "в синхронизированный");
        Caller ob3 = new Caller(target, "мир!");

        // ожидание завершения потока
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Прервано");
        }
    }
}
```

Вот вывод этой программы.

```
Добро пожаловать[в синхронизированный[мир!]
]
]
```

Как видите, вызывая метод `sleep()`, метод `call()` позволяет переключиться на выполнение другого потока. Это приводит к смешанному выводу трех строк сообщений. В этой программе нет ничего, что предотвращает вызов потоками одного и того же метода в одном и том же объекте в одно и то же время. Это называется состоянием *гонок* (race condition) или конфликтом, поскольку три потока соревнуются друг с другом в окончании выполнения метода. Этот пример использует метод `sleep()`, чтобы сделать эффект повторяемым и наглядным. В большинстве ситуаций этот эффект менее заметен и менее предсказуем, поскольку вы не можете предвидеть, когда произойдет переключение контекста. Это может привести к тому, что программа один раз отработает правильно, а другой раз — нет.

Чтобы исправить эту программу, следует *сериализировать* доступ к методу `call()`. То есть в одно и то же время вы должны разрешить доступ к этому методу только одному потоку. Чтобы сделать это, вам нужно просто предварить объявление метода `call()` ключевым словом `synchronized`, как показано ниже.

```
class Callme {
    synchronized void call(String msg) {
    ...
```

Это предотвратит доступ другим потокам к методу `call()`, когда один из них уже использует его. После того как слово `synchronized` добавлено к методу `call()`, результат работы программы будет выглядеть следующим образом.

```
[Добро пожаловать]
[в синхронизированный]
[мир!]
```

Всякий раз, когда у вас есть метод или группа методов, которые манипулируют внутренним состоянием объекта в многопоточной среде, следует использовать ключевое слово synchronized, чтобы исключить ситуацию с гонками. Помните, что как только поток входит в любой синхронизированный метод экземпляра, ни один другой поток не может войти ни в один синхронизированный метод того же экземпляра. Однако несинхронизированные методы экземпляра по-прежнему остаются доступными для вызова.

Оператор synchronized

Хотя создание синхронизированных методов в ваших классах — простой и эффективный способ синхронизации, все же он работает не во всех случаях. Чтобы понять, почему, рассмотрим следующее. Предположим, что вы хотите синхронизировать доступ к объектам классов, которые не были предназначены для многопоточного доступа. То есть класс не использует синхронизированных методов. Более того, класс был написан не вами, а независимым разработчиком, и у вас нет доступа к его исходному коду. Значит, вы не можете добавить слово synchronized к объявлению соответствующих методов класса. Как может быть синхронизирован доступ к объектам такого класса? К счастью, существует довольно простое решение этой проблемы: вы просто заключаете вызовы методов этого класса в блок synchronized.

Вот общая форма оператора synchronized.

```
synchronized(объект) {
    // операторы, подлежащие синхронизации
}
```

Здесь *объект* — это ссылка на синхронизируемый объект. Блок synchronized гарантирует, что вызов метода объекта произойдет только тогда, когда текущий поток успешно войдет в монитор объекта.

Ниже показана альтернативная версия предыдущего примера с использованием синхронизированного блока внутри метода run().

```
// Эта программа использует синхронизированный блок.
class Callme {
    void call(String msg) {
        System.out.print("[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        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);
        t.start();
    }
```

```
    // синхронизированные вызовы call()
    public void run() {
        synchronized(target) { // синхронизированный блок
            target.call(msg);
        }
    }
}

class Synch1 {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Добро пожаловать");
        Caller ob2 = new Caller(target, "в синхронизированный");
        Caller ob3 = new Caller(target, "мир!");

        // ожидание завершения потока
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Прервано");
        }
    }
}
```

Здесь метод `call()` не модифицирован словом `synchronized`. Вместо этого используется оператор `synchronized` внутри метода `run()` класса `Caller`. Это позволяет получить тот же корректный результат, что и в предыдущем примере, поскольку каждый поток ожидает окончания выполнения своего предшественника.

Межпотоковые коммуникации

Предыдущие примеры, безусловно, блокировали другие потоки от асинхронного доступа к некоторым методам. Это использование неявных мониторов объектов Java является мощным средством, но вы можете достичь более подробного уровня контроля за счет межпроцессных коммуникаций. Как вы увидите, это особенно просто в Java.

Как обсуждалось ранее, многопоточность заменила программирование на основе циклов событий за счет разделения задач на дискретные, логически обособленные единицы. Потоки предоставляют также второе преимущество: они исключают опрос. Опрос обычно реализуется в виде цикла, используемого для периодической проверки некоторого условия. Как только условие истинно, выполняется определенное действие. Это расходует время процессора. Например, рассмотрим классическую проблему, когда один поток создает некоторые данные, а другой принимает их. Чтобы сделать проблему интересней, предположим, что поставщик данных должен ожидать, когда потребитель завершит работу, прежде чем поставщик создаст новые данные. В системах с опросом потребитель данных тратит много циклов процессора на ожидание данных от поставщика. Как только поставщик завершает работу, он должен начать опрос, расходующий циклы процессора в ожидании завершения работы потребителя данных, и т.д. Понятно, что такая ситуация нежелательна.

Чтобы избежать опроса, Java включает элегантный механизм межпроцессных коммуникаций с использованием методов `wait()`, `notify()` и `notifyAll()`. Эти методы реализованы как финальные в классе `Object`, поэтому они доступны

всем классам. Все три метода могут быть вызваны только из синхронизированного контекста. Хотя с точки зрения компьютерной науки они концептуально сложны, правила применения этих методов достаточно просты.

- Метод `wait()` принуждает вызывающий поток отдать монитор и приостановить выполнение до тех пор, пока какой-нибудь другой поток не войдет в тот же монитор и не вызовет метод `notify()`.
- Метод `notify()` возобновляет работу потока, который вызвал метод `wait()` в том же объекте.
- Метод `notifyAll()` возобновляет работу всех потоков, которые вызвали метод `wait()` в том же объекте. Одному из потоков дается доступ.

Эти методы объявлены в классе `Object`, как показано ниже.

```
final void wait() throws InterruptedException
final void notify()
final void notifyAll()
```

Существуют дополнительные формы метода `wait()`, позволяющие указать время ожидания.

Прежде чем рассматривать пример, демонстрирующий межпотоковое взаимодействие, необходимо сделать одно важное замечание. Хотя метод `wait()` обычно ожидает до тех пор, пока не будет вызван метод `notify()` или `notifyAll()`, существует вероятность, что в очень редких случаях ожидающий поток может быть возобновлен поддельным сигналом. При этом ожидающий поток возобновляется без вызова метода `notify()` или `notifyAll()`. (По сути, поток возобновляется без явных причин.) Из-за этой маловероятной возможности Oracle рекомендует выполнять вызовы метода `wait()` внутри цикла, проверяющего условие, по которому поток ожидает. В приведенном ниже примере показан такой подход.

А пока рассмотрим пример, использующий методы `wait()` и `notify()`. Для начала проанализируем следующий простой пример программы, некорректно реализующий задачу "поставщик/потребитель". Она состоит из четырех классов: `Q` — очередь, которую нужно синхронизировать, `Producer` — объект-поток, который создает элементы очереди, `Consumer` — объект-поток, принимающий элементы очереди, и `PC` — крошечный класс, который создает объекты классов `Q`, `Producer` и `Consumer`.

```
// Неправильная реализация поставщика 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;

    Producer(Q q) {
        this.q = q;
```

```
        new Thread(this, "Поставщик").start();
    }

    public void run() {
        int i = 0;

        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;

    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Потребитель").start();
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PC {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);

        System.out.println("Для останова нажмите Control-C.");
    }
}
```

Несмотря на то что методы `put()` и `get()` в классе `Q` синхронизированы, ничто не остановит переполнение потребителя поставщиком, как и ничто не помешает потребителю извлечь один и тот же компонент очереди дважды. То есть вы получите неверный результат, показанный ниже (точная последовательность может быть другой, в зависимости от скорости процессора и загрузки).

```
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 того, как поставщик отправляет 1, запускается потребитель и получает это же значение 1 пять раз подряд. Затем поставщик продолжает работу и поставляет значения от 2 до 7, не давая возможности потребителю получить их.

Правильный способ написания этой программы на языке Java заключается в том, чтобы применить методы `wait()` и `notify()` для передачи сигналов в обоих направлениях.

```
// Правильная реализация поставщика и потребителя.
class Q {
    int n;
    boolean valueSet = false;

    synchronized int get() {
        while(!valueSet)
        try {
            wait();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException перехвачено");
        }

        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("InterruptedException перехвачено");
            }

            this.n = n;
            valueSet = true;
            System.out.println("Отправлено: " + n);
            notify();
        }
    }

    class Producer implements Runnable {
        Q q;

        Producer(Q q) {
            this.q = q;
            new Thread(this, "Поставщик").start();
        }

        public void run() {
            int i = 0;

            while(true) {
                q.put(i++);
            }
        }
    }

    class Consumer implements Runnable {
        Q q;

        Consumer(Q q) {
            this.q = q;
            new Thread(this, "Потребитель").start();
```

```
        }

        public void run() {
            while(true) {
            q.get();
        }
    }
}

class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);

        System.out.println("Для останова нажмите Control-C.");
    }
}
```

Внутри метода `get()` вызывается метод `wait()`. Это приостанавливает работу потока до тех пор, пока объект класса `Producer` не известит вас о том, что данные прочитаны. Когда это происходит, выполнение внутри метода `get()` продолжается. После получения данных метод `get()` вызывает метод `notify()`. Это сообщает объекту класса `Producer`, что все в порядке и можно помещать в очередь следующий элемент данных. Внутри метода `put()` метод `wait()` приостанавливает выполнение до тех пор, пока объект класса `Consumer` не извлечет элемент из очереди. Когда выполнение возобновится, следующий элемент данных помещается в очередь и вызывается метод `notify()`. Это сообщает объекту класса `Consumer`, что он теперь может извлечь его.

Ниже приведен вывод программы, который доказывает, что теперь синхронизация работает корректно.

```
Отправлено: 1
Получено: 1
Отправлено: 2
Получено: 2
Отправлено: 3
Получено: 3
Отправлено: 4
Получено: 4
Отправлено: 5
Получено: 5
```

Взаимная блокировка

Следует избегать особого типа ошибок, имеющего отношение к многозадачности. Это — взаимная блокировка (deadlock), которая происходит, когда потоки имеют циклическую зависимость от пары синхронизированных объектов. Предположим, что один поток входит в монитор объекта `X`, а другой — в монитор объекта `Y`. Если поток в объекте `X` попытается вызвать любой синхронизированный метод объекта `Y`, он будет блокирован, как и ожидалось. Однако если поток объекта `Y`, в свою очередь, попытается вызвать любой синхронизированный метод объекта `X`, то поток будет ожидать вечно, поскольку для получения доступа к объекту `X` он должен снять свой собственный блок с объекта `Y`, чтобы первый поток мог работать. Взаимная блокировка является ошибкой, которую трудно отладить, по двум следующим причинам.

- В общем, она случается довольно редко, когда выполнение двух потоков точно совпадает по времени.
- Она может происходить, когда в этом участвует более двух потоков и двух синхронизированных объектов. (То есть взаимная блокировка может случиться в результате более сложной последовательности событий, чем в приведенном примере.)

Чтобы полностью разобраться с этим явлением, лучше рассмотреть его в действии. Следующий пример создает два класса `A` и `B` с методами `foo()` и `bar()` соответственно, которые приостанавливаются непосредственно перед попыткой вызова метода другого класса. Главный класс, названный `Deadlock`, создает экземпляры классов `A` и `B`, а затем запускает второй поток, устанавливающий состояние взаимной блокировки. Методы `foo()` и `bar()` используют метод `sleep()`, чтобы стимулировать появление взаимной блокировки.

```
// Пример взаимной блокировки.
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 A.last");
    }
}

class Deadlock implements Runnable {
    A a = new A();
    B b = new B();
```

```
    Deadlock() {
        Thread.currentThread().setName("MainThread");
        Thread t = new Thread(this, "RacingThread");
        t.start();

        a.foo(b); // получить блокировку внутри этого потока.
        System.out.println("Назад в главный поток");
    }

    public void run() {
        b.bar(a); // получить блокировку b в другом потоке.
        System.out.println("Назад в другой поток");
    }

    public static void main(String args[]) {
        new Deadlock();
    }
}
```

Когда вы запустите эту программу, то увидите следующий результат.

```
MainThread вошел в A.foo
RacingThread вошел в B.bar
MainThread пытается вызвать B.last()
RacingThread пытается вызвать A.last()
```

Поскольку эта программа заблокирована, вам придется нажать <Ctrl+C> для завершения программы. Вы можете видеть весь поток и дамп кеша монитора, нажав <Ctrl+Break>. Вы увидите, что `RacingThread` владеет монитором на `b`, в то время как последний ожидает монитора на `a`. В то же время `MainThread` владеет `a` и ожидает `b`. Эта программа никогда не завершится. Как иллюстрирует этот пример, если ваша многопоточная программа неожиданно зависла, то первое, что следует проверить, — возможность взаимной блокировки.

Приостановка, возобновление и останов потоков

Иногда возникает необходимость в приостановке выполнения потоков. Например, отдельный поток может использоваться для отображения времени дня. Если пользователю не нужны часы, то этот поток можно приостановить. В любом случае приостановка потока — простая вещь. Выполнение приостановленного потока может быть легко возобновлено.

Механизм временной либо окончательной остановки потока, а также его возобновления отличался в ранних версиях Java, таких как Java 1.0, от современных версий, начиная с Java 2. Хотя при написании нового кода нужно придерживаться нового подхода, вы по-прежнему должны понимать, как эти операции были реализованы в ранних версиях среды Java. Например, может возникнуть необходимость в поддержке или обновлении старого, унаследованного, кода. Вам также может понадобиться понять, почему в этот механизм были внесены изменения. По этим причинам в следующем разделе описан изначальный способ управления выполнением потоков, а за ним следует раздел, описывающий, как это реализовано в новых версиях.

Приостановка, возобновление и останов потоков в Java 1.1 и более ранних версиях

До версии Java 2 программы использовали методы `suspend()` и `resume()`, определенные в классе `Thread` для приостановки и возобновления потоков. Они имеют следующую форму.

```
final void suspend()
final void resume()
```

Хотя использовать эти методы больше не рекомендуется, в следующей программе демонстрируется их применение, чтобы вы могли понять, как они работали.

```
// Использование методов suspend() и resume() только в демонстрационных
// целях. Для нового кода они не рекомендуются.
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);
        t.start(); // запуск потока
    }

    // Точка входа потока.
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(200);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " прерван.");
        }
        System.out.println(name + " завершен.");
    }
}

class SuspendResume {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("Один");
        NewThread ob2 = new NewThread("Два");

        try {
            Thread.sleep(1000);
            ob1.t.suspend();
            System.out.println("Приостановка потока Один");
            Thread.sleep(1000);
            ob1.t.resume();
            System.out.println("Возобновление потока Один");
            ob2.t.suspend();
            System.out.println("Приостановка потока Два");
            Thread.sleep(1000);
            ob2.t.resume();
            System.out.println("Возобновление потока Два");
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван");
        }
```

```
        // Ожидание 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 завершен.");
    }
}
```

Пример вывода этой программы показан ниже (в вашем случае он может отличаться, в зависимости от скорости и загрузки процессора).

```
Новый поток: Thread[Один,5,main]
Один: 15
Новый поток: Thread[Два,5,main]
Два: 15
Один: 14
Два: 14
Один: 13
Два: 13
Один: 12
Два: 12
Один: 11
Два: 11
Приостановка потока Один
Два: 10
Два: 9
Два: 8
Два: 7
Два: 6
Возобновление потока Один
Приостановка потока Два
Один: 10
Один: 9
Один: 8
Один: 7
Один: 6
Возобновление потока Two
Ожидание завершения потоков.
Два: 5
Один: 5
Два: 4
Один: 4
Два: 3
Один: 3
Два: 2
Один: 2
Два: 1
Один: 1
Два завершен.
Один завершен.
Главный поток завершен.
```

Класс `Thread` также определяет метод `stop()`, который останавливает поток. Его сигнатура такова.

```
final void stop()
```

Остановленный поток уже не может быть возобновлен с помощью метода `resume()`.

Современный способ приостановки, возобновления и остановки потоков

Хотя применение методов `suspend()`, `resume()` и `stop()` класса `Thread` выглядит как исключительно разумный и удобный подход к управлению выполнением потоков, они не должны использоваться в новых программах Java. И вот почему. Метод `suspend()` класса `Thread` несколько лет назад был объявлен нежелательным в Java 2. Это было сделано потому, что иногда он способен порождать серьезные системные сбои. Предположим, что поток пытается получить блокировки на критичных структурах данных. Если поток приостановить в этот момент, блокировки не будут установлены. Другие потоки, которые ожидают эти ресурсы, могут оказаться взаимно блокированными.

Метод `resume()` также нежелателен. Он не вызовет проблем, но не может быть использован без метода `suspend()` как своего дополнения.

Метод `stop()` класса `Thread` также объявлен устаревшим в Java 2. Это было сделано потому, что он также иногда может послужить причиной серьезных системных сбоев. Предположим, что поток выполняет запись в критически важную структуру данных и успел выполнить только частичное обновление. Если его остановить в этот момент, структура данных может оказаться в поврежденном состоянии.

Поскольку вы не можете использовать методы `suspend()`, `resume()` или `stop()` для управления потоками, то можете подумать, что теперь вообще нет способа приостановить, возобновить или прервать поток. К счастью, это не так. Вместо этого поток должен быть спроектирован так, чтобы метод `run()` периодически проверял, должно ли выполнение потока быть приостановлено, возобновлено или прервано. Обычно для этого используется переменная-флаг, указывающая состояние потока. До тех пор, пока этот флаг имеет значение "запущен", метод `run()` должен продолжать выполнение. Если флаг имеет значение "прерван", поток должен приостановиться. Если флаг получает значение "стоп", то поток должен завершиться. Конечно, существует множество способов написать такой код, но основной принцип остается неизменным для всех программ.

В следующем примере показано, как методы `wait()` и `notify()`, унаследованные от класса `Object`, могут применяться для управления выполнением потока. Этот пример похож на программу из предыдущего раздела. Однако вызовы устаревших методов здесь исключены. Рассмотрим работу этой программы.

Класс `NewThread` содержит переменную экземпляра типа `boolean` по имени `suspendFlag`, используемую для управления выполнением потока. Конструктор инициализирует ее значением `false`. Метод `run()` содержит блок `synchronized`, который проверяет состояние переменной `suspendFlag`. Если ее значение равно `true`, вызывается метод `wait()` для приостановки выполнения потока. Метод `mysuspend()` устанавливает значение переменной `suspendFlag` в состояние `true`. Метод `myresime()` устанавливает значение переменной `suspendFlag` в состояние `false` и вызывает метод `notify()`, чтобы "разбудить" поток. И наконец, метод `main()` модифицирован для вызова методов `mysuspend()` и `myresime()`.

```
// Приостановка и возобновление потока 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;
        t.start(); // запустить поток
    }

    // Точка входа 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 + " завершен.");
    }

    synchronized void mysuspend() {
        suspendFlag = true;
    }

    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}

class SuspendResume {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("Один");
        NewThread ob2 = new NewThread("Два");

        try {
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Приостановка потока Один");
            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Возобновление потока Один");
            ob2.mysuspend();
            System.out.println("Приостановка потока Два");
            Thread.sleep(1000);
            ob2.myresume();
            System.out.println("Возобновление потока Два");
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван");
        }

        // ожидание 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 завершен");
    }
}
```

Вывод этой программы идентичен приведенному в предыдущем разделе. Чуть позднее в этой книге вы найдете еще примеры, в которых используется современный механизм управления потоками. Хотя этот метод не так "чист", как старый, его следует придерживаться, дабы избежать ошибок времени выполнения. Это — подход, который *должен* применяться во всем новом коде.

Получение состояния потока

Как уже упоминалось в этой главе, поток может находиться в нескольких разных состояниях. Вы можете получить текущее состояние потока, вызвав метод `getState()`, определенный в классе `Thread` следующим образом.

```
Thread.State getState()
```

Метод возвращает значение типа `Thread.State`, указывающее состояние потока на момент вызова. Перечисление `State` определено в классе `Thread`. (Перечисление — это список именованных констант; подробно он обсуждается в главе 12.) Значения, которые может возвратить метод `getState()`, перечислены в табл. 11.2.

Схема на рис. 11.1 демонстрирует взаимоотношения между различными состояниями потока.

С учетом наличия экземпляра класса `Thread`, вы можете использовать метод `getState()`, чтобы получить состояние потока. Например, следующий код определяет, находится ли поток по имени `thrd` в состоянии `RUNNABLE` во время вызова метода `getState()`.

```
Thread.State ts = thrd.getState();
if(ts == Thread.State.RUNNABLE) // ...
```

Таблица 11.2. Возвращаемые значения метода `getState()`

Значения	Состояние
`BLOCKED`	Поток приостановил выполнение, поскольку ожидает получения блокировки
`NEW`	Поток еще не начал выполнение
`RUNNABLE`	Поток в настоящее время выполняется или начнет выполняться, когда получит доступ к процессору
`TERMINATED`	Поток закончил выполнение
`TIMED_WAITING`	Поток приостановил выполнение на определенный промежуток времени, например, после вызова метода `sleep()`. Поток переходит также в это состояние при вызове метода `wait()` или `join()`
`WAITING`	Поток приостановил выполнение, поскольку он ожидает некое действие. Например, вызова версий методов `wait()` или `join()`, прекращающих ожидание

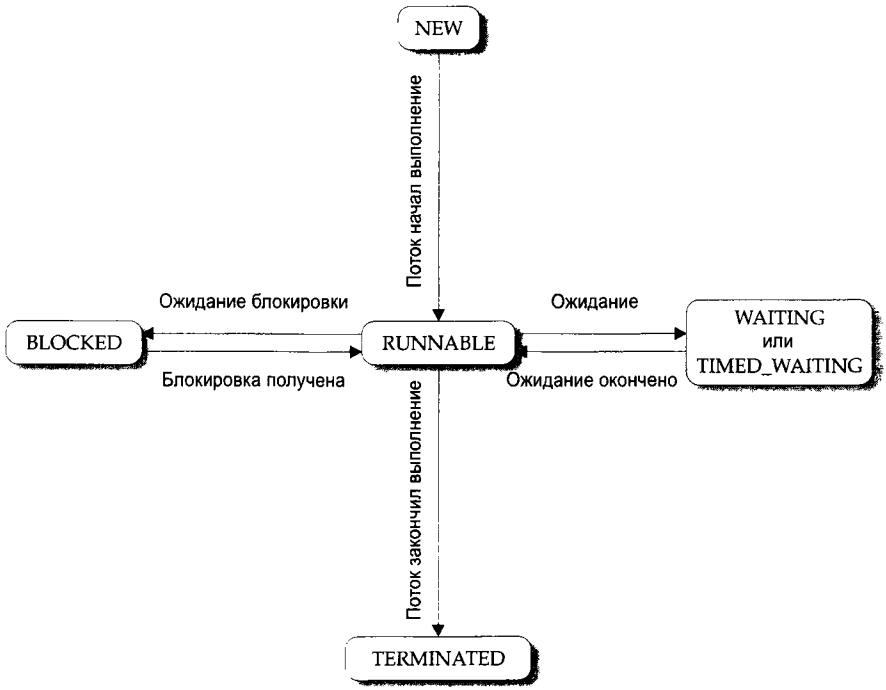


Рис. 11.1. Состояния потока

Важно понять, что состояние потока может измениться после вызова метода `getState()`. Таким образом, в зависимости от обстоятельств, состояние, полученное при вызове метода `getState()`, мгновение спустя может не отражать фактическое состояние потока. По этой и другим причинам метод `getState()` не предназначен для использования средствами синхронизации потоков. Он, прежде всего, используется при отладке или для профилирования характеристик потока во время выполнения.

Использование многопоточности

Чтобы эффективно использовать многопоточные средства Java вы должны научиться мыслить "параллельно", а не "последовательно". Например, когда вы имеете две подсистемы в программе, которые могут выполняться одновременно, оформите их в виде отдельных потоков. При взвешенном применении многопоточности вы будете писать очень эффективные программы. Однако следует проявлять осторожность. Если вы создадите слишком много потоков, можете даже снизить производительность всей программы, вместо того чтобы повысить ее. Помните, что переключение контекстов между потоками требует определенных накладных расходов. Если вы создадите очень много потоков, больше времени процессора будет затрачено на переключение контекста, нежели на само выполнение программы! Последний момент: чтобы создать приложение с интенсивными вычислениями, допускающее автоматическое масштабирование для использования доступных процессоров в многоядерной системе, используйте новую инфраструктуру Fork/Join Framework, описанную в главе 27.

ГЛАВА 12

Перечисления, автоупаковка и аннотации (метаданные)

В настоящей главе рассматриваются три относительно новых дополнения к языку Java: перечисления, автоупаковка и аннотации (называемые также метаданными). Каждое из них увеличивает мощь языка, предлагая изящный подход к решению часто возникающих задач программирования. В главе также обсуждаются оболочки типов Java и рефлексия.

Перечисления

Версиям языка Java, предшествовавшим JDK 5, недоставало одного средства, необходимость в котором чувствовали многие программисты, — перечисления. В простейшей форме *перечисление* — это список именованных констант. Хотя Java включает и другие средства, имеющие похожую функциональность, такие как финальные переменные, многим программистам все же не хватало концептуальной чистоты перечислений — в особенности потому, что они применяются во многих других языках программирования. Начиная с JDK 5 перечисления были добавлены к языку Java и, наконец, стали доступны программистам.

В простейшей форме перечисления Java подобны перечислениям в других языках. Однако это сходство поверхностно. В языках вроде C++ перечисления просто представляют собой списки целочисленных констант. В языке Java перечисления определяют тип класса. За счет реализации перечислений в виде классов сама концепция перечисления значительно расширяется. Например, в языке Java перечисления могут иметь конструкторы, методы и переменные экземпляра. Таким образом, хотя воплощения перечислений пришлось ждать несколько лет, реализация их в языке Java стоила того.

Основные понятия о перечислениях

Перечисления создаются с использованием ключевого слова `enum`. Например, ниже показано простое перечисление сортов яблок.

```
// Перечисление сортов яблок.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

Идентификаторы `Jonathan`, `GoldenDel` и так далее называются *константами перечисления*. Каждая из них явно объявлена как открытый статический финальный член класса `Apple`. Более того, их тип — это тип перечисления, в котором они

объявлены; в данном случае это Apple. То есть в языке Java эти константы называются *самотипизированными*, причем *само* относится к окружающему перечислению.

Объявив перечисление, вы можете создавать переменные этого типа. Однако даже несмотря на то, что перечисления определяют тип класса, вы не можете создавать объекты этого типа с помощью оператора new. Вместо этого вы объявляете и используете переменную перечисления почти таким же образом, как это делается с элементарными типами. Например, ниже объявляется переменная ap перечислимого типа Apple.

```
Apple ap;
```

Поскольку переменная ap имеет тип Apple, присвоить ей можно только те значения, которые определены в перечислении. Например, здесь переменной ap присваивается значение RedDel.

```
ap = Apple.RedDel;
```

Обратите внимание на то, что значению RedDel предшествует тип Apple.

Две перечислимые константы можно проверять на равенство с помощью оператора отношения ==. Например, следующий оператор сравнивает переменную ap с константой Apple.GoldenDel.

```
if (ap == Apple.GoldenDel) // ...
```

Перечислимые значения также могут быть использованы в управляющей конструкции switch. Конечно же, все операторы case должны использовать константы из того же перечисления enum, что и выражение switch. Например, следующий оператор switch абсолютно корректен.

```
// Использование enum для управления switch.
switch(ap) {
case Jonathan:
    // ...
case Winesap:
    // ...
```

Обратите внимание на то, что в операторах case имена перечислимых констант используются без квалифицированного имени их типа перечисления. То есть применяется Winesap, а не Apple.Winesap. Дело в том, что тип перечисления в операторе switch уже неявно задает тип перечисления для операторов case. Нет необходимости квалифицировать константы в операторах case именем типа их перечисления. Фактически попытка сделать это приведет к ошибке компиляции.

Когда константа перечисления отображается, скажем, методом println(), выводится ее имя. Например, следующая строка кода

```
System.out.println(Apple.Winesapp)
```

отобразит имя Winesapp.

Приведенная ниже программа собирает все вместе и демонстрирует применение перечисления Apple.

```
// Перечисление сортов яблок.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

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

```
        Apple ap;

        ap = Apple.RedDel;

        // Вывод значения enum.
        System.out.println("Значение ap: " + ap);
        System.out.println();

        ap = Apple.GoldenDel;

        // Сравнение двух перечислимых значений.
        if(ap == Apple.GoldenDel)
            System.out.println("ap содержит GoldenDel.\n");

        // Применение enum для управления оператором switch.
        switch(ap) {
            case Jonathan:
                System.out.println("Jonathan красный.");
                break;
            case GoldenDel:
                System.out.println("Golden Delicious желтый.");
                break;
            case RedDel:
                System.out.println("Red Delicious красный.");
                break;
            case Winesap:
                System.out.println("Winesap красный.");
                break;
            case Cortland:
                System.out.println("Cortland красный.");
                break;
        }
    }
}
```

Эта программа создает следующий вывод.

```
Значение ap: RedDel
ap содержит GoldenDel.
Golden Delicious желтый.
```

Методы `values()` и `valueOf()`

Перечисления автоматически включают два предопределенных метода: `values()` и `valueOf()`. Их общая форма выглядит так.

```
public static тип_перечисления [] values()
public static тип_перечисления valueOf(String строка)
```

Метод `values()` возвращает массив, содержащий список констант перечисления. Метод `valueOf()` возвращает константу перечисления, значение которой соответствует строке, переданной в аргументе *строка*. В обоих случаях *тип_перечисления* — это тип перечисления. Например, в случае с перечислением `Apple`, показанным выше, типом возвращаемого значения `Apple.valueOf("Winesapp")` будет `Winesapp`.

В следующей программе демонстрируется применение методов `values()` и `valueOf()`.

```
// Использование встроенных методов перечислений.

// Перечисление сортов яблок.
enum Apple {
```

```
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
    public static void main(String args[])
    {

        Apple ap;

        System.out.println("Константы Apple:");

        // применение values()
        Apple allapples[] = Apple.values();
        for(Apple a : allapples)
            System.out.println(a);

        System.out.println();

        // применение valueOf()
        ap = Apple.valueOf("Winesap");
        System.out.println("ap содержит " + ap);
    }
}
```

Вывод этой программы таков.

```
Константы Apple:
Jonathan
GoldenDel
RedDel
Winesap
Cortland
ap содержит Winesap
```

Обратите внимание на то, что программа использует стиль "for-each" цикла `for` для перебора массива констант, возвращенных методом `values()`. В целях демонстрации создается переменная `allapples`, которой присваивается ссылка на массив перечислимых значений. Но это не обязательно, поскольку цикл `for` можно написать, как показано ниже, избежав применения переменной `allapples`.

```
for(Apple a : Apple.values())
    System.out.println(a);
```

Обратите также внимание на то, как значение, соответствующее имени `Winesapp`, получается вызовом метода `valueOf()`.

```
ap = Apple.valueOf("Winesap");
```

Как объяснялось ранее, метод `valueOf()` возвращает перечислимое значение, ассоциированное с именем константы, переданным в строке.

На заметку! Программисты на C/C++ обратят внимание на то, что в языке Java значительно упрощено преобразование между читабельной для человека формой константы перечисления и ее бинарным значением. Это существенное преимущество подхода к перечислениям языка Java.

Перечисления в Java являются типами классов

Как уже объяснялось, перечисление в Java — это тип класса. Хотя вы не можете создать экземпляр перечисления с помощью оператора `new`, в остальном перечисление обладает всеми возможностями, которые имеются у других классов.

Тот факт, что перечисление определяет класс, придает такую мощь перечислениям Java, которой лишены перечисления в других языках. Например, вы можете предоставлять им конструкторы, добавлять переменные экземпляров и методы и даже реализовывать интерфейсы.

Важно понимать, что каждая константа перечисления является объектом своего класса перечисления. То есть, когда вы определяете конструктор для перечисления, он вызывается при каждом создании константы перечисления. Также каждая константа перечисления имеет свою собственную копию переменных экземпляра, объявленных перечислением. Например, рассмотрим следующую версию перечисления `Apple`.

```
// Использование конструктора enum, переменной экземпляра и метода.
enum Apple {
    Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

    private int price; // цена каждого яблока

    // Конструктор
    Apple(int p) { price = p; }

    int getPrice() { return price; }
}

class EnumDemo3 {
    public static void main(String args[])
    {
        Apple ap;

        // Отобразить цену Winesap.
        System.out.println("Winesap стоит " +
                           Apple.Winesap.getPrice() +
                           " центов.\n");

        // Отобразить цены всех сортов яблок.
        System.out.println("Все цены яблок:");
        for(Apple a : Apple.values())
            System.out.println(a + " стоит " + a.getPrice() +
                               " центов.");
    }
}
```

Ниже показан вывод программы.

```
Winesap стоит 15 центов.
Все цены яблок:
Jonathan стоит 10 центов.
GoldenDel стоит 9 центов.
RedDel стоит 12 центов.
Winesap стоит 15 центов.
Cortland стоит 8 центов.
```

Данная версия перечисления `Apple` добавляет следующее. Первое — это переменная экземпляра `price`, которая применяется для хранения цены каждого сорта яблок. Второе — конструктор `Apple()`, которому передается цена яблок. Третье — метод `getPrice()`, возвращающий значение цены.

Когда в методе `main()` объявляется переменная `ap`, конструктор `Apple()` вызывается однажды для каждой объявленной константы. Следует отметить, что аргументы конструктору передаются помещением их в скобки после каждой константы.

```
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
```

Эти переменные передаются параметру p конструктора Apple(), который затем присваивает их переменной экземпляра price. Опять же конструктор вызывается однажды для каждой константы.

Поскольку каждая константа перечисления имеет собственную копию переменной экземпляра price, вы можете получить цену определенного сорта яблок вызовом метода getPrice(). Например, цену сорта Winesap в методе main() получим следующим вызовом.

```
Apple.Winesap.getPrice()
```

Цены всех сортов получим при переборе перечисления в цикле for. Поскольку копия переменной экземпляра price существует для каждой перечислимой константы, значение, ассоциированное с одной константой, отделено и отличается от значения, ассоциированного с другой константой. Это мощная концепция, которая доступна только в случае реализации перечислений в виде классов, как это сделано в языке Java.

Хотя предыдущий пример содержит только один конструктор, перечисление может представлять две или более перегруженных форм, как это может делать любой другой класс. Например, приведенная ниже версия перечисления Apple предлагает конструктор по умолчанию, инициализирующий цену значением -1, означающим, что цена не указана.

```
// Использование конструкторов enum.
enum Apple {
    Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);

    private int price; // цена каждого яблока

    // Конструктор
    Apple(int p) { price = p; }

    // Перегруженный конструктор
    Apple() { price = -1; }

    int getPrice() { return price; }
}
```

Обратите внимание на то, что в этой версии константе RedDel не передается аргумент. Это означает, что вызывается конструктор по умолчанию и переменная цены RedDel устанавливается равной -1.

Здесь есть два ограничения относительно перечислений. Во-первых, перечисление не может наследоваться от другого класса. Во-вторых, перечисление не может быть суперклассом. Это значит, что перечисление не может быть расширено. Во всем остальном перечисление ведет себя, как любой другой тип класса. Ключевой момент – помнить, что каждая константа перечисления является объектом класса, в котором она определена.

Перечисления наследуются от класса Enum

Хотя вы не можете наследовать суперкласс при объявлении перечисления, все перечисления автоматически наследуют класс java.lang.Enum. Этот класс определяет несколько методов, доступных к использованию всеми перечислениями. Класс Enum подробно рассматривается в части II, но три его метода требуют описания прямо сейчас.

Вы можете получить значение, которое указывает позицию константы в списке констант перечисления. Это называется *порядковым значением* (ordinal value), которое извлекается с помощью вызова метода `ordinal()`, показанного ниже.

```
final int ordinal()
```

Этот метод возвращает порядковое значение вызывающей константы. Порядковые значения начинаются с нуля. То есть в перечислении `Apple` константа `Johnatan` имеет порядковое значение 0, константа `GoldenDel` — 1, константа `RedDel` — 2 и т.д.

Вы можете сравнить порядковые значения двух констант одного и того же перечисления с помощью метода `compareTo()`. Он имеет следующую общую форму.

```
final int compareTo(тип_перечисления e)
```

Здесь *тип_перечисления* — тип перечисления, а *e* — константа, которую нужно сравнить с вызывающей константой. Помните, что вызывающая константа и *e* должны относиться к одному перечислению. Если вызывающая константа имеет порядковое значение меньше чем *e*, то метод `compareTo()` возвращает отрицательное значение. Если два порядковых значения одинаковы, возвращается нуль. Если вызывающая константа имеет порядковое значение больше чем *e*, то возвращается положительное значение.

Вы можете сравнить на эквивалентность перечислимую константу с любым другим объектом, используя метод `equals()` — переопределенный метод `equals()` класса `Object`. Хотя метод `equals()` может сравнивать перечислимые константы с любым другим объектом, эти два объекта будут эквивалентны только в случае, если оба являются ссылкой на одну и ту же константу из одного и того же перечисления. Простое совпадение порядковых значений не заставит метод `equals()` вернуть значение `true`, если две константы принадлежат разным перечислениям.

Помните, что вы можете сравнивать две ссылки перечислений на эквивалентность, используя оператор `==`.

В следующей программе демонстрируется применение методов `ordinal()`, `compareTo()` и `equals()`.

```
// Демонстрация 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 значения с помощью ordinal().
        System.out.println("Вот все константы " +
                           " и их порядковые значения: ");
        for(Apple a : Apple.values())
            System.out.println(a + " " + a.ordinal());

        ap = Apple.RedDel;
        ap2 = Apple.GoldenDel;
        ap3 = Apple.RedDel;

        System.out.println();

        // Демонстрация compareTo() и equals()
```

```
        if(ap.compareTo(ap2) < 0)
            System.out.println(ap + " идет перед " + ap2);

        if(ap.compareTo(ap2) > 0)
            System.out.println(ap2 + " идет перед " + ap);

        if(ap.compareTo(ap3) == 0)
            System.out.println(ap + " эквивалентно " + ap3);

        System.out.println();

        if(ap.equals(ap2))
            System.out.println("Error!");

        if(ap.equals(ap3))
            System.out.println(ap + " equals " + ap3);

        if(ap == ap3)
            System.out.println(ap + " == " + ap3);
    }
}
```

Ниже показан вывод этой программы.

```
Вот все константы и их порядковые значения:
Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4

GoldenDel идет перед RedDel
RedDel эквивалентно RedDel

RedDel эквивалентно RedDel
RedDel == RedDel
```

Еще один пример перечисления

Прежде чем двигаться дальше, рассмотрим еще один пример применения перечисления. В главе 9 создавалась программа для автоматического принятия решений. В этой версии переменные NO, YES, MAYBE, LATER, SOON и NEVER были объявлены в интерфейсе и использованы для представления возможных ответов. Хотя в таком подходе нет ничего технологически неверного, применение перечислений — более подходящее решение. Здесь представлена усовершенствованная версия этой программы, которая использует перечисление по имени `Answers` для представления ответов. Можете сравнить эту версию с оригинальной из главы 9.

```
// Усовершенствованная версия программы принятия решений
// из главы 9. В этой версии для представления
// используется enum, а не переменные экземпляра.

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("Позднее");
                break;
            case SOON:
                System.out.println("Вскоре");
                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 вы знаете, в языке Java для хранения базовых типов данных, поддерживаемых языком, используются элементарные типы (также называемые простыми типами), такие как `int` или `double`. Элементарные типы, в отличие от объектов, используются для таких значений из соображений производительности.

Применение объектов для этих значений приводит к нежелательным накладным расходам, даже в случае простейших вычислений. Поэтому элементарные типы не являются частью иерархии объектов и не наследуются от класса `Object`.

Несмотря на то что элементарные типы обеспечивают выигрыш производительности, бывают случаи, когда вам может понадобиться объектное представление. Например, вы не можете передать в метод элементарный тип по ссылке. Кроме того, многие стандартные структуры данных, реализованные в языке Java, оперируют объектами, а это означает, что вы не можете применять эти структуры данных для сохранения элементарных типов. Чтобы справиться с такими (и подобными) ситуациями, язык Java предлагает *оболочки типов*, которые представляют собой классы, заключающие элементарный тип в объект. Классы-оболочки типов детально описаны в части II, но поскольку они имеют непосредственное отношение к автоупаковке Java, кратко рассмотрим их здесь.

Оболочки типов — это `Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`, `Character` и `Boolean`. Эти классы предоставляют широкий диапазон методов, позволяющий в полной мере интегрировать элементарные типы в иерархию объектных типов Java. Каждый из них кратко рассматривается далее.

Класс `Character`

Этот класс является оболочкой вокруг типа `char`. Конструктор `Character()` выглядит следующим образом.

```
Character(char символ)
```

Здесь *символ* указывает символ, который будет помещен в оболочку при создании объекта класса `Character`.

Чтобы получить значение типа `char`, содержащееся в объекте класса `Character`, вызовите метод `charValue()`, показанный ниже.

```
char charValue()
```

Он возвращает инкапсулированный символ.

Класс `Boolean`

Это — оболочка вокруг значений типа `boolean`. В ней определены следующие конструкторы.

```
Boolean(boolean логическоеЗначение)
Boolean(String логическаяСтрока)
```

В первой версии *логическоеЗначение* должно быть либо `true`, либо `false`. Во второй версии, если *логическаяСтрока* содержит строку `"true"` (в верхнем или нижнем регистре), новый объект класса `Boolean` будет содержать значение `true`. В противном случае он будет содержать значение `false`.

Чтобы получить значение типа `boolean` из объекта класса `Boolean`, используйте метод `booleanValue()`, показанный ниже.

```
boolean booleanValue()
```

Он возвращает эквивалент типа `boolean` вызывающего объекта.

Оболочки числовых типов

До сих пор наиболее часто используемыми оболочками типов являются те, что представляют числовые значения. Это `Byte`, `Short`, `Integer`, `Long`, `Float` и `Double`. Все оболочки числовых типов наследуют абстрактный класс `Number`.

Этот класс объявляет методы, которые возвращают значение объекта в каждом из числовых форматов. Вот эти методы.

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

Например, метод `doubleValue()` возвращает значение объекта как тип `double`, метод `floatValue()` — как тип `float` и т.д. Эти методы реализованы каждой из оболочек числовых типов.

Все оболочки числовых типов определяют конструкторы, которые позволяют создавать объекты из заданного значения или строкового представления этого значения. Например, вот как выглядят конструкторы для класса `Integer`.

```
Integer(int число)
Integer(String строка)
```

Если *строка* не содержит числового значения, то передается исключение `NumberFormatException`.

Все оболочки типов переопределяют метод `toString()`. Этот метод возвращает читабельную для человека форму значения, содержащегося в оболочке. Например, это позволяет выводить значение, передавая объект оболочки типа методу `println()` без необходимости преобразования его в элементарный тип. В следующей программе показано, как использовать оболочку числового типа для инкапсуляции значения и последующего его извлечения.

```
// Демонстрация оболочки типа.
class Wrap {
    public static void main(String args[]) {

        Integer iOb = new Integer(100);

        int i = iOb.intValue();

        System.out.println(i + " " + iOb); // отображает 100 100
    }
}
```

Эта программа помещает целое значение 100 внутрь объекта класса `Integer` по имени `iOb`. Затем она получает значение за счет вызова метода `intValue()` и помещает результат в переменную `i`.

Процесс инкапсуляции значения в объект называется *упаковкой* (boxing). То есть следующая строка программы упаковывает значение 100 в объект класса `Integer`.

```
Integer iOb = new Integer(100);
```

Процесс извлечения значения из оболочки типа называется *распаковкой* (unboxing). Например, приведенная строка программы распаковывает значение объекта `iOb`.

```
int i = iOb.intValue();
```

Та же общая процедура, что используется в предыдущей программе для упаковки и распаковки значений, применялась и в исходной версии языка Java. Однако в версию Java J2SE 5 были внесены фундаментальные усовершенствования за счет добавления автоматической упаковки, описанной ниже.

Автоупаковка

Начиная с JDK 5 в язык Java добавлены два важных средства: *автоупаковка* (autoboxing) и *автораспаковка* (autounboxing). Автоупаковка – это процесс, в результате которого элементарный тип автоматически инкапсулируется (упаковывается) в эквивалентную ему оболочку типа всякий раз, когда требуется объект этого типа. Нет необходимости явного создания объекта. Автораспаковка – это процесс автоматического извлечения значения упакованного объекта (распаковка) из оболочки типа, когда нужно получить его значение. Нет необходимости вызывать методы вроде `intValue()` или `doubleValue()`.

Добавление автоматической упаковки и распаковки значительно упрощает реализацию некоторых алгоритмов, исключая необходимость в ручной упаковке и распаковке значений. Это также помогает предотвратить ошибки. Более того, это очень важно для средства обобщения классов и алгоритмов, которые оперируют только объектами. И наконец, автоупаковка существенно облегчает работу с инфраструктурой коллекций Collection Framework, описанной в части II.

С применением автоупаковки больше нет необходимости в ручном создании объектов для оболочки элементарных типов. Вам нужно только присвоить значение ссылке оболочки типов. Язык Java автоматически создаст эти объекты для вас. Например, вот современный способ создания объекта класса `Integer`, который содержит значение 100.

```
Integer iOb = 100; // автоупаковка int
```

Обратите внимание: объект не создается явно оператором `new`. Язык Java делает это за вас автоматически.

Чтобы распаковать объект, просто присваивайте ссылку на объект переменной элементарного типа. Например, чтобы распаковать объект `iOb`, следует использовать следующую строку.

```
int i = iOb; // автораспаковка
```

Java справляется с деталями за вас.

Вот предыдущая программа, переписанная для использования автоупаковки и автораспаковки.

```
// Демонстрация автоупаковки/автораспаковки.
class AutoBox {
    public static void main(String args[]) {

        Integer iOb = 100; // автоупаковка int

        int i = iOb;       // автораспаковка

        System.out.println(i + " " + iOb); // отображает 100 100
    }
}
```

Автоупаковка и методы

В дополнение к простым случаям присвоения, автоупаковка происходит автоматически всякий раз, когда элементарный тип должен быть преобразован в объект. Автораспаковка происходит всякий раз, когда объект должен быть преобразован в элементарный тип. Таким образом, автоупаковка и автораспаковка могут

осуществляться, когда аргумент передается методу либо когда значение возвращается из метода. Рассмотрим пример.

```
// Автоупаковка/автораспаковка происходит
// с методами параметров и возвращаемыми значениями.

class AutoBox2 {
    // принять параметр Integer и вернуть
    // значение int;
    static int m(Integer v) {
        return v ; // автораспаковка int
    }

    public static void main(String args[]) {
        // Передача int методу m() и присвоение возвращаемого значения
        // объекту Integer. Здесь аргумент 100 автоматически
        // упаковывается в Integer. Возвращаемое значение также
        // упаковывается в Integer.
        Integer iOb = m(100);

        System.out.println(iOb);
    }
}
```

Эта программа отображает следующий результат.

```
100
```

Обратите внимание на то, что в этой программе метод `m()` задает параметр класса `Integer` и возвращает результат типа `int`. Внутри метода `main()` методу `m()` передается значение 100. Поскольку метод `m()` ожидает объект класса `Integer`, это значение автоматически упаковывается. Затем метод `m()` возвращает эквивалент аргумента типа `int`. Это заставляет автоматически упаковаться в переменную `v`. Далее это значение типа `int` присваивается объекту `iOb` в методе `main()`, что вызывает автоматическую упаковку результата типа `int`.

Автоупаковка и распаковка в выражениях

Вообще, автоупаковка и распаковка происходят всякий раз, когда требуется преобразование в объект или из объекта. Это касается выражений. Внутри выражения числовой объект автоматически распаковывается. Выходной результат выражения при необходимости упаковывается заново. Например, рассмотрим следующую программу.

```
// Автоупаковка/распаковка происходят в выражениях.

class AutoBox3 {
    public static void main(String args[]) {

        Integer iOb, iOb2;
        int i;

        iOb = 100;
        System.out.println("Исходное значение iOb: " + iOb);

        // Следующее автоматически распаковывает iOb,
        // выполняет его приращение, затем повторно
        // упаковывает результат обратно в iOb.
        ++iOb;
        System.out.println("После ++iOb: " + iOb);
```

```
        // Здесь iOb распаковано, выражение вычисляется,
        // а результат снова упаковывается и
        // присваивается iOb2.
        iOb2 = iOb + (iOb / 3);
        System.out.println("iOb2 после выражения: " + iOb2);

        // Вычисляется то же самое выражение,
        // но результат не упаковывается.
        i = iOb + (iOb / 3);
        System.out.println("i после выражения: " + i);
    }
}
```

Вывод показан ниже.

```
Исходное значение iOb: 100
После ++iOb: 101
iOb2 после выражения: 134
i после выражения: 134
```

Обратите особое внимание на следующую строку программы.

```
++iOb;
```

Она увеличивает на 1 значение объекта `iOb`. Это работает следующим образом: объект `iOb` распаковывается, значение увеличивается и результат упаковывается вновь.

Автоматическая распаковка также позволяет смешивать разные типы числовых объектов в одном выражении. Как только значение распаковано, применяются стандартные правила повышения типов и преобразования. Например, следующая программа абсолютно корректна.

```
class AutoBox4 {
    public static void main(String args[]) {

        Integer iOb = 100;
        Double dOb = 98.6;

        dOb = dOb + iOb;
        System.out.println("dOb после выражения: " + dOb);
    }
}
```

Результат показан ниже.

```
dOb после выражения: 198.6
```

Как видите, оба объекта — и `dOb` класса `Double`, и `iOb` класса `Integer` — участвуют в сложении, и результат повторно упаковывается и сохраняется в объекте `dOb`.

Благодаря автоупаковке, можно применять целочисленные объекты для управления оператором `switch`. Например, рассмотрим 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");
}
```

Когда вычисляется выражение switch, объект iOb распаковывается и возвращается его значение типа int.

Как показывают примеры программ, благодаря автоупаковке и распаковке, применение числовых объектов в выражениях интуитивно понятно и просто. В прошлом такой код требовал применения приведений и вызовов методов вроде intValue().

Автоупаковка и распаковка значений классов Boolean и Character

Как описывалось ранее, Java также поддерживает оболочки для типов boolean и char — соответственно классы Boolean и Character. Автоупаковка/распаковка также применима к этим типам. Например, рассмотрим следующую программу.

```
// Автоупаковка/распаковка Boolean и Character.

class AutoBox5 {
    public static void main(String args[]) {

        // Автоупаковка/распаковка boolean.
        Boolean b = true;

        // b автоматически распаковывается
        // при использовании в условном выражении if.
        if(b) System.out.println("b равна true");

        // Автоупаковка/распаковка char.
        Character ch = 'x'; // упаковка char
        char ch2 = ch;      // распаковка char

        System.out.println("ch2 равна " + ch2);
    }
}
```

Результат этой программы таков.

```
b равна true
ch2 равна x
```

Наиболее важный момент в этой программе, о котором стоит упомянуть, — это автоматическая распаковка объекта b внутри условного выражения if. Как вы помните, управляющее условие выражения if при вычислении должно возвращать значение типа boolean. Благодаря автораспаковке, значение типа boolean, содержащееся в объекте b, автоматически распаковывается при вычислении условного выражения. То есть с появлением автоупаковки и распаковки стало возможным применять объекты класса Boolean для управления в операторе if.

Благодаря автоупаковке и распаковке, объект класса Boolean теперь также может применяться для управления всеми циклическими конструкциями Java. Когда объект класса Boolean применяется в качестве условия в выражении while, for или do/while, он автоматически распаковывается в свой эквивалент типа boolean. Например, вот новый допустимый код.

```
Boolean b;
// ...
while(b) { // ...
```

Автоупаковка и распаковка помогают предотвратить ошибки

В дополнение к удобству, которое они предоставляют, автоупаковка и распаковка могут также помочь избежать ошибок. Например, рассмотрим следующую программу.

```
// Ошибка, порожденная "ручной" распаковкой.
class UnboxingError {
    public static void main(String args[]) {

        Integer iOb = 1000;       // автоупаковка значения 1000

        int i = iOb.byteValue(); // ручная распаковка, как byte !!!

        System.out.println(i);    // не отображает 1000 !
    }
}
```

Эта программа отображает не ожидаемое значение 1000, а –24! Причина в том, что значение внутри объекта `iOb` распаковано вручную вызовом метода `byteVal()`, что привело к усечению значения 1000, хранящегося в объекте `iOb`. В результате получилось "мусорное" значение –24, которое было присвоено переменной `i`. Авторапаковка предотвращает этот тип ошибок, поскольку значение объекта `iOb` всегда будет автоматически распаковываться в значение, совместимое с типом `int`.

В общем, поскольку автоупаковка всегда создает правильный объект, а авторапаковка всегда порождает правильное значение, нет опасности получить неверное значение или неверный объект. В тех редких случаях, когда вам нужно получить значение типа, отличающегося от того, который создается автоматически, вы можете вручную упаковывать и распаковывать значения. Конечно, выгоды от автоупаковки/авторапаковки в этом случае теряются. В принципе, новый код должен использовать автоупаковку/авторапаковку. Это — правильный способ написания кода на Java.

Предостережения

Теперь, когда Java включает средства автоматической упаковки/распаковки. некоторые могут подумать, что соблазнительно применять исключительно объекты классов вроде `Integer` или `Double`, исключая использование элементарных типов. Например, теперь можно написать код вроде следующего.

```
// Плохое применение автоупаковки/авторапаковки!
Double a, b, c;

a = 10.0;
b = 4.0;

c = Math.sqrt(a*a + b*b);

System.out.println("Гипотенуза равна " + c);
```

В этом примере объекты класса `Double` хранят значения, которые используются для вычисления гипотенузы прямоугольного треугольника. Несмотря на то что этот код технически корректен и работает правильно, все же это очень плохое использование автоупаковки/авторапаковки. Он намного менее эффективен, чем эквивалентный код, использующий элементарный тип `double`. Причина в том,

что автоупаковка и автораспаковка добавляют накладные расходы, которые отсутствуют в случае применения элементарных типов.

Использование оболочек типов следует ограничивать только теми случаями, когда требуется объектное представление элементарных типов. Автоупаковка и автораспаковка были добавлены в Java вовсе не в качестве обходного маневра для исключения элементарных типов.

Аннотации (метаданные)

Начиная с JDK 5 язык Java поддерживает средство, которое позволяет встроить информацию поддержки в исходные файлы. Эта информация, называемая также *аннотацией*, не меняет действия программы. То есть аннотация сохраняет семантику программ неизменной. Однако эта информация может быть использована различными инструментальными средствами как во время разработки, так и в период развертывания. Например, аннотация может обрабатываться генераторами исходного кода. Термин *метаданные* также используется для именования этого средства, но более описательный термин *средства аннотирования программ* применяется значительно шире.

Основы аннотирования

Аннотации создаются с использованием механизма, основанного на интерфейсе. Начнем с примера. Вот объявление аннотации под названием `MyAnno`.

```
// Простой тип аннотации.
@interface MyAnno {
    String str();
    int val();
}
```

Обратите внимание на символ `@`, предшествующий ключевому слову `interface`. Это указывает компилятору, что объявлен тип аннотации. Далее отметим два метода — `str()` и `val()`. Все аннотации состоят только из объявлений методов. Однако тела этих методов вы не определяете. Вместо этого их реализует Java. Более того, методы ведут себя в большей степени подобно полям, как вы вскоре убедитесь.

Аннотация не может включать слова `extends`. Однако все аннотации автоматически расширяют интерфейс `Annotation`. То есть интерфейс `Annotation` является суперинтерфейсом для всех аннотаций. Он объявлен в пакете `java.lang.annotation`. В интерфейсе `Annotation` переопределены методы `hashCode()`, `equals()` и `toString()`, которые определены в классе `Object`. В нем также задан метод `annotationType()`, возвращающий объект класса `Class`, который представляет вызывающую аннотацию.

Объявив аннотацию, вы можете использовать ее для аннотирования объявления. Любой тип объявления может иметь аннотацию, ассоциированную с ним. Например, аннотироваться могут классы, методы, поля, параметры и константы перечислений. Аннотированной может быть даже сама аннотация. Во всех случаях аннотация предшествует остальной части объявления.

Когда вы применяете аннотацию, то присваиваете значения ее членам. Например, ниже показан вариант применения аннотации `MyAnno` к объявлению метода.

```
// Аннотирование метода.
@MyAnno(str = "Пример аннотации", val = 100)
public static void myMeth() { // ...
```

Эта аннотация связана с методом myMeth(). Посмотрите внимательно на ее синтаксис. За именем аннотации, которому предшествует @, следует взятый в скобки список инициализаторов членов. Чтобы присвоить значение члену, оно присваивается его имени. Таким образом, в этом примере строка "Пример аннотации" присваивается члену str объекта аннотации MyAnno. Обратите внимание на то, что в этом присваивании никаких скобок за str не следует. Когда члену аннотации присваивается значение, используется только его имя. То есть члены аннотации в данном контексте выглядят как поля.

Политика удержания аннотации

Прежде чем объяснять аннотации дальше, необходимо обсудить *политику удержания аннотаций* (annotation retention policies). Политика удержания определяет, в какой точке аннотация отбрасывается. Java определяет три такие политики, которые инкапсулированы в перечислении java.lang.annotation.RetentionPolicy. Это SOURCE, CLASS и RUNTIME.

Аннотации с политикой удержания SOURCE содержатся только в исходном файле и отбрасываются при компиляции.

Аннотации с политикой удержания CLASS сохраняются в файле .class во время компиляции. Однако они недоступны JVM во время выполнения.

Аннотации с политикой удержания RUNTIME сохраняются в файле .class во время компиляции и остаются доступными JVM во время выполнения. То есть политика RUNTIME предоставляет аннотации наиболее высокую степень постоянства.

На заметку! Аннотация объявлений локальных переменных в файле .class не сохраняется.

Политика удержания для аннотации задается с помощью одной из встроенных аннотаций Java: @Retention. Ее общая форма показана ниже.

@Retention(*политика_удержания*)

Здесь *политика_удержания* должна быть одной из описанных ранее констант. Если для аннотации не указано никакой политики удержания, используется политика удержания CLASS.

В следующем примере аннотации MyAnno устанавливается политика удержания RUNTIME с помощью аннотации @Retention. То есть аннотация MyAnno будет доступна JVM во время выполнения программы.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}
```

Получение аннотаций во время выполнения с использованием рефлексии

Хотя аннотации спроектированы, в основном, для использования инструментами разработки и развертывания, если они задают политику удержания RUNTIME, то могут быть опрошены во время выполнения любой программой Java за счет использования рефлексии. Рефлексия – это средство, позволяющее получить информацию о классе во время выполнения программы. Программный интерфейс (API) рефлексии содержится в пакете java.lang.reflect. Существует множество спо-

собов применения рефлексии, и мы не будем здесь обсуждать их все. Тем не менее мы пройдемся по нескольким примерам, имеющим отношение к аннотациям.

Первый шаг в использовании рефлексии — это получение объекта класса `Class`, представляющего класс, аннотацию которого нужно получить. Класс `Class` — это один из встроенных классов Java, определенный в пакете `java.lang`. Он детально рассматривается в части II. Есть разные способы получения объекта класса `Class`. Один из простейших — вызвать метод `getClass()`, определенный в классе `Object`. Его общая форма показана ниже.

```
final Class<?> getClass()
```

Метод возвращает объект класса `Class`, который представляет вызывающий объект.

На заметку! Обратите внимание на символы `<?>`, следующие за словом `Class` в объявлении метода `getClass()` выше. Это связано со средствами обобщений Java. Метод `getClass()` и несколько других связанных с рефлексией методов, обсуждаемых в этой главе, используют обобщения. Обобщения описаны в главе 14. Однако понимание обобщений не обязательно, чтобы уяснить фундаментальные принципы рефлексии.

После того как получите объект класса `Class`, вы можете использовать его методы для получения информации о различных элементах, объявленных в классе, включая его аннотацию. Если хотите получить аннотации, ассоциированные с определенным элементом класса, следует сначала получить объект, представляющий этот элемент. Например, класс `Class` представляет (помимо прочих) методы `getMethod()`, `getField()` и `getConstructor()`, которые возвращают информацию о методе, поле и конструкторе соответственно. Эти методы возвращают объекты классов `Method`, `Field` и `Constructor`.

Чтобы понять этот процесс, рассмотрим пример, который получает аннотации, ассоциированные с методом. Для этого вы сначала получаете объект класса `Class`, представляющий класс, затем вызываете метод `getMethod()` этого объекта, указав имя метода. Метод `getMethod()` имеет следующую общую форму.

```
Method getMethod(String имяМетода, Class<?> ... типыПараметров)
```

Имя метода передается в параметре *имяМетода*. Если метод принимает аргументы, то объекты класса `Class`, представляющие их типы, также должны быть указаны в списке *типыПараметров*. Обратите внимание на то, что *типыПараметров* — это список аргументов переменной длины (vararg). Это означает, что вы можете задать столько типов параметров, сколько нужно, включая нуль. Метод `getMethod()` возвращает объект класса `Method`, который представляет метод. Если метод не может быть найден, передается исключение `NoSuchMethodException`.

От объектов классов `Class`, `Method`, `Field` или `Constructor` вы можете получить специфические аннотации, ассоциированные с этим объектом, обратившись к методу `getAnnotation()`. Его общая форма представлена ниже.

```
<A extends Annotation> getAnnotation(Class<A> типАннотации)
```

Здесь *типАннотации* — это объект класса `Class`, представляющий аннотацию, в которой вы заинтересованы. Метод возвращает ссылку на аннотацию. Используя эту ссылку, вы можете получить значения, ассоциированные с членами аннотации.

Метод возвращает значение `null`, если аннотация не найдена; это случай, когда аннотация не имеет аннотации `@Retention`, устанавливающей политику удержания `RUNTIME`. Ниже показана программа, которая собирает все описанные части вместе и использует рефлексию для отображения аннотации, ассоциированной с методом.

```
import java.lang.annotation.*;
import java.lang.reflect.*;
```

```
// Объявление типа аннотации.
@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 из метода
        // и отобразить значения членов.
        try {
            // Для начала получить Class,
            // представляющий класс.
            Class<?> c = ob.getClass();

            // Теперь получить объект Method,
            // представляющий этот метод.
            Method m = c.getMethod("myMeth");

            // Далее получить аннотацию класса.
            MyAnno anno = m.getAnnotation(MyAnno.class);

            // Наконец, отобразить аннотацию.
            System.out.println(anno.str() + " " + anno.val());
        } catch (NoSuchMethodException exc) {
            System.out.println("Метод не найден.");
        }
    }

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

Вот как выглядит результат работы этой программы.

```
Пример аннотации 100
```

Эта программа использует рефлексию, как описано, чтобы получить и отобразить значения переменных `str` и `val` аннотации `MyAnno`, ассоциированной с методом `myMeth()` в классе `Meta`. Есть несколько моментов, на которые следует обратить особое внимание. Первый момент — выражение `MyAnno.class` в строке.

```
MyAnno anno = m.getAnnotation(MyAnno.class);
```

Это выражение вычисляется как объект `Class` типа `MyAnno` — аннотация. Это называется *литералом класса*. Вы можете использовать выражения этого типа всякий раз, когда требуется объект `Class` известного класса. Например, следующий оператор служит для получения объекта `Class` для класса `Meta`.

```
Class<?> c = Meta.class;
```

Конечно, такой подход работает, только когда вы знаете имя класса объекта заранее, что не всегда возможно. Вообще, вы можете получать литерал класса для классов, интерфейсов, элементарных типов и массивов. (Помните, что синтаксис `<?>` имеет отношение к средствам обобщений Java; они описаны в главе 14.)

Второй интересный момент — это способ получения значений, ассоциированных с переменными `str` и `val`, когда они выводятся в следующей строке.

```
System.out.println(anno.str() + " " + anno.val());
```

Обратите внимание на то, что они вызываются с применением синтаксиса вызова методов. Тот же подход используется всякий раз, когда требуется получить член аннотации.

Второй пример применения рефлексии

В предыдущем примере метод `myMeth()` не имел параметров. Другими словами, когда вызывался метод `getMethod()`, передавалось только имя `myMeth`. Однако для того, чтобы получить метод, который имеет параметры, следует задать объекты класса, представляющие типы этих параметров, в виде аргументов метода `getMethod()`. Например, ниже показана слегка измененная версия предыдущей программы.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class Meta {

    // myMeth теперь имеет 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);
    }
}
```

Результат работы этой версии будет таким.

```
Два параметра 19
```

В этой версии метод `myMeth()` принимает параметры типа `String` и `int`. Чтобы получить информацию об этом методе, метод `getMethod()` должен быть вызван следующим образом.

```
Method m = c.getMethod("myMeth", String.class, int.class);
```

Здесь объекты `Class`, представляющие типы `String` и `int`, передаются в виде дополнительных аргументов.

Получение всех аннотаций

Вызвав метод `getAnnotations()` с позицией, вы можете получить сразу все аннотации, имеющие аннотацию `@Retention`, с установленной политикой удержания `RUNTIME`, которые ассоциированы с этой позицией. Общая форма этого метода выглядит так.

```
Annotation[] getAnnotations()
```

Этот метод возвращает массив аннотаций. Метод `getAnnotations()` может быть вызван для объектов классов `Class`, `Method`, `Constructor` и `Field`.

Вот еще один пример с рефлексией, который показывает, как получить все аннотации, ассоциированные с классом и методом. Он объявляет две аннотации. Затем он использует их для аннотирования класса и метода.

```
// Показать все аннотации 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 тестового класса")
@MyAnno(str = "Meta2", val = 99)
class Meta2 {

    @What(description = "Аннотация тестового 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 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 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();
        }
}
```

Ниже показан результат работы этой программы.

```
Все аннотации для Meta2:
@What(description=Аннотация тестового класса)
@MyAnno(str=Meta2, val=99)

Все аннотации для myMeth:
@What(description=Аннотация тестового метода)
@MyAnno(str=Testing, val=100)
```

Эта программа использует метод `getAnnotations()` для получения массива всех аннотаций, ассоциированных с классом `Meta2` и методом `myMeth()`. Как объяснялось, метод `getAnnotations()` возвращает массив объектов интерфейса `Annotation`. Вспомните, что `Annotation` — это суперинтерфейс для всех интерфейсов аннотаций и что он переопределяет метод `toString()` класса `Object`. То есть когда выводится ссылка на интерфейс `Annotation`, вызывается его метод `toString()` для создания строки, описывающей аннотацию, что и демонстрирует предыдущий пример.

Интерфейс `AnnotatedElement`

Методы `getAnnotation()` и `getAnnotations()`, использованные в предыдущем примере, определены интерфейсом `AnnotatedElement`, который определен в пакете `java.lang.reflect`. Этот интерфейс поддерживает рефлексию для аннотации и реализован классами `Method`, `Field`, `Constructor`, `Class` и `Package`.

В дополнение к методам `getAnnotation()` и `getAnnotations()`, интерфейс `AnnotatedElement` определяет два других метода. Первый из них — метод `getDeclaredAnnotations()`, который имеет следующую общую форму.

```
Annotation[] getDeclaredAnnotations()
```

Этот метод возвращает неунаследованные аннотации, представленные в вызывающем объекте. Второй — это метод `isAnnotationPresent()`, имеющий такую форму.

```
boolean isAnnotationPresent(Class<? extends Annotation> типАннотации)
```

Он возвращает значение `true`, если аннотация, заданная в *типАннотации*, ассоциирована с вызывающим объектом. В противном случае метод возвращает значение `false`.

Использование значений по умолчанию

Вы можете придать членам аннотации значения по умолчанию, которые будут использоваться, когда применяется аннотация. Значение по умолчанию указывается добавлением ключевого слова `default` к объявлению члена. Это выглядит следующим образом.

```
тип member() default значение;
```

Здесь *значение* должно иметь тип, совместимый с *тип*. Вот как выглядит аннотация `@MyAnno`, переписанная с использованием значений по умолчанию.

```
// Объявление типа аннотации, включающее значения по умолчанию.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str() default "Тестирование";
    int val() default 9000;
}
```

Это объявление определяет значение по умолчанию `"Тестирование"` для переменной-члена `str` и `9000` — для переменной-члена `val`. Это означает, что ни одно из значений не обязательно указывать при использовании аннотации `@MyAnno`. Однако любому из них или обоим сразу можно присвоить значение при необходимости. Таким образом, существует четыре способа применения аннотации `@MyAnno`.

```
@MyAnno() // значения str и val принимаются по умолчанию
@MyAnno(str = "некоторая строка") // val — по умолчанию
@MyAnno(val = 100)                // str — по умолчанию
@MyAnno(str = "Тестирование", val = 100) // нет умолчаний
```

В следующей программе демонстрируется использование значений по умолчанию в аннотациях.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Объявление типа аннотаций с включением значений по умолчанию.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str() default "Тестирование";
    int val() default 9000;
}

class Meta3 {

    // Аннотирование метода с использованием значений по умолчанию.
    @MyAnno()
    public static void myMeth() {
        Meta3 ob = new Meta3();

        // Получить аннотацию к методу
        // и отобразить 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();
    }
}
```

Вывод будет таким.

```
Тестирование 9000
```

Аннотация-маркер

Это — специальный вид аннотаций, которые не содержат членов. Единственное назначение аннотации-маркера — пометить (маркировать) объявление. То есть его присутствие как аннотации существенно. Лучший способ определить, имеется ли аннотация-маркер, — воспользоваться методом `isAnnotationPresent()`, который определен в интерфейсе `AnnotatedElement`.

Рассмотрим пример использования аннотации-маркера. Поскольку такая аннотация не имеет членов, следует просто определить, присутствует она или нет.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Аннотация-маркер.
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }

class Marker {
    // Аннотирование метода с помощью маркера.
    // Обратите внимание на необходимость скобок ().
    @MyMarker
    public static void myMeth() {
        Marker ob = new Marker();

        try {
            Method m = ob.getClass().getMethod("myMeth");

            // Определение наличия аннотации.
            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();
    }
}
```

Показанный ниже вывод подтверждает наличие аннотации `@MyMarker`.

```
MyMarker присутствует.
```

Обратите внимание на то, что нет необходимости после аннотации `@MyMarker` указывать скобки. То есть аннотация `@MyMarker` применяется просто с использованием ее имени.

```
@MyMarker
```

Не будет ошибкой указать пустые скобки, однако в этом нет необходимости.

Одночленные аннотации

Одночленная аннотация содержит, как должно быть понятно, только один член. Она работает подобно обычной аннотации, за исключением того, что допускает сокращенную форму указания значения члена. Когда присутствует только один член, вы можете просто задать его значение; когда аннотация применяется, вы не обязаны указывать имя члена. Однако для того, чтобы использовать это сокращение, член должен иметь имя `value`.

Ниже показан пример создания и использования одночленной аннотации.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Одночленная аннотация.
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
    int value(); // именем переменной должно быть value
}

class Single {

    // Аннотирование метода одночленной аннотацией.
    @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()); // отображает 100

        } catch (NoSuchMethodException exc) {
            System.out.println("Метод не найден.");
        }
    }

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

Как и ожидалось, эта программа отображает значение 100. Здесь аннотация `@MySingle` применяется для аннотирования метода `myMeth()`, как показано ниже.

```
@MySingle(100)
```

Обратите внимание на то, что не обязательно указывать `value =`.

Вы можете применять синтаксис одночленных аннотаций и при использовании аннотаций с другими членами, но все остальные члены должны иметь значения по умолчанию. Например, ниже добавляется член `xyz` со значением по умолчанию, равным 0.

```
@interface SomeAnno {
    int value();
    int xyz() default 0;
}
```

В случаях, когда вы хотите использовать значение по умолчанию для члена `xyz`, можете применить аннотацию `@SomeAnno`, как показано ниже, просто указав значение для `value` с использованием синтаксиса одночленных аннотаций.

```
@SomeAnno(88)
```

В этом случае член `xyz` по умолчанию принимает значение 0, а член `value` — 88. Конечно, чтобы задать другое значение для члена `xyz`, необходимо, чтобы оба члена были инициированы явно.

```
@SomeAnno(value = 88, xyz = 99)
```

Помните, что когда вы применяете одночленные аннотации, именем члена должно быть `value`.

Встроенные аннотации

В Java определено очень много встроенных аннотаций. Большинство из них специализированы, но восемь имеют общее назначение. Четыре из них импортируются из пакета `java.lang.annotation`: `@Retention`, `@Documented`, `@Target` и `@Inherited`. Четыре другие аннотации — `@Override`, `@Deprecated`, `@SafeVarargs` и `@SuppressWarnings` — включены в пакет `java.lang`. Каждая из них описана ниже.

Аннотация `@Retention`

Эта аннотация предназначена для применения только в качестве аннотации к другим аннотациям. Определяет политику удержания, как было описано в настоящей главе.

Аннотация `@Documented`

Это — маркер-интерфейс, который сообщает инструменту, что аннотация должна быть документирована. Он предназначен для использования только в качестве аннотации к объявлению аннотации.

Аннотация `@Target`

Эта аннотация задает типы объявлений, к которым может быть применима аннотация. Предназначена для использования только в качестве аннотации к другим аннотациям. Аннотация `@Target` принимает один аргумент, который должен быть константой из перечисления `ElementType`. Этот аргумент задает типы объявлений, к которым может быть применена аннотация. Эти константы описаны в табл. 12.1 вместе с типами объявлений, к которым они относятся.

Таблица 12.1. Константы из перечисления `ElementType`

Целевая константа	Аннотация может быть применена к
`ANNOTATION_TYPE`	Другой аннотации
`CONSTRUCTOR`	Конструктору
`FIELD`	Полю
`LOCAL_VARIABLE`	Локальной переменной
`METHOD`	Методу
`PACKAGE`	Пакету
`PARAMETER`	Параметру
`TYPE`	Классу, интерфейсу или перечислению

Вы можете задать одно или несколько этих значений в аннотации `@Target`. Чтобы указать множественные значения, следует поместить их внутрь ограниченного фигурными скобками списка. Например, чтобы указать, что аннотация применима только к полям и локальным переменным, нужно использовать следующую аннотацию `@Target`.

```
@Target( { ElementType.FIELD, ElementType.LOCAL_VARIABLE } )
```

Аннотация `@Inherited`

Это — аннотация-маркер, которая может применяться в другом объявлении аннотации. Более того, она касается только тех аннотаций, что будут использоваться в объявлениях классов. Аннотация `@Inherited` позволяет аннотации суперкласса быть

унаследованной в подклассе. Таким образом, когда осуществляется запрос к подклассу на предмет специфической аннотации, то, если этой аннотации в подклассе нет, проверяется суперкласс. Если запрошенная аннотация присутствует у суперкласса и она аннотирована как `@Inherited`, то эта аннотация будет возвращена.

Аннотация `@Override`

Аннотация `@Override` — аннотация-маркер, которая может применяться только в методах. Метод, аннотированный как `@Override`, должен переопределять метод суперкласса. Если он этого не делает, в результате происходит ошибка времени компиляции. Она используется для обеспечения того, что метод суперкласса будет действительно переопределен, а не просто перегружен.

Аннотация `@Deprecated`

Эта аннотация-маркер указывает, что объявление устарело и должно быть заменено более новой формой.

Аннотация `@SafeVarargs`

Аннотация `@SafeVarargs` — аннотация-маркер, применяемая к методам и конструкторам. Она указывает, что никакие небезопасные действия, связанные с параметром переменного количества аргументов, недопустимы. Она используется для подавления неотмеченных предупреждений в остальном безопасном коде относительно нессылочных типов с переменным количеством аргументов и параметрическим созданием экземпляра массива. (Нессылочный тип — это, по существу, обобщенный тип, описанный в главе 14.) Применяется только к методам или конструкторам с переменным количеством аргументов, которые объявлены как `static` или `final`. Добавлено в JDK 7.

Аннотация `@SuppressWarnings`

Эта аннотация указывает, что одно или более предупреждений, которые могут быть выданы компилятором, следует подавить. Подавляемые предупреждения задаются именами в строковой форме. Эта аннотация может быть применима к объявлениям любого типа.

Некоторые ограничения

Существует некоторое количество ограничений, касающихся объявления аннотаций. Во-первых, одна аннотация не может наследовать другую. Во-вторых, все методы, объявленные в аннотации, не должны принимать параметров. Более того, они должны возвращать один из перечисленных ниже типов:

- элементарный тип, такой как `int` или `double`;
- объект класса `String` или `Class`;
- тип перечисления;
- тип другой аннотации;
- массив одного из предыдущих типов.

Аннотации не могут быть обобщенными. Другими словами, они не могут принимать параметры типа. (Обобщения рассматриваются в главе 14.) И наконец, в методах аннотации не может быть указана конструкция `throws`.

ГЛАВА 13 Ввод-вывод, аплеты и другие темы

Настоящая глава посвящена двум наиболее важным пакетам Java: `io` и `applet`. Пакет `io` поддерживает базовую систему ввода-вывода Java, включая файловый ввод-вывод. Пакет `applet` поддерживает аплеты. Поддержка ввода-вывода и аплетов осуществляется ядром библиотек программного интерфейса (API), а не ключевыми словами языка. По этой причине углубленное обсуждение этих тем содержится в части II, где рассматриваются классы API. В этой главе описаны основы этих двух подсистем, чтобы вы смогли увидеть, как они интегрированы в язык Java и встроены в общий контекст программирования на языке Java и его исполняющей системы. В этой главе также рассматриваются новый оператор JDK 7 *try-с-ресурсами* и последние из ключевых слов Java: `transient`, `volatile`, `instanceof`, `native`, `strictfp` и `assert`. Завершает главу описание статического импорта и применения ключевого слова `this`.

Основы ввода-вывода

Как вы могли заметить, читая главу 12, до сих пор в примерах программ было задействовано не так много операций ввода-вывода. Фактически, помимо методов `print()` и `println()`, никаких методов ввода-вывода, в общем-то, и не применялось. Причина этого проста: большинство реальных приложений Java не являются текстовыми консольными программами. Вместо этого они являются программами с графическим пользовательским интерфейсом на базе библиотек Abstract Window Toolkit (AWT) и Swing или веб-приложениями. Хотя текстовые, консольные программы великолепны в качестве учебных примеров, они не занимают сколько-нибудь значительную часть в мире реальных программ. К тому же, поддержка консольного ввода-вывода в Java ограничена и не слишком удобна в использовании — даже для простейших программ. Тем не менее текстовый консольный ввод-вывод вполне применим в реальном программировании на языке Java.

Язык Java обеспечивает мощную и гибкую поддержку ввода-вывода, когда это касается файлов и сетей. Система ввода-вывода Java целостна и последовательна. Фактически, если однажды разобраться с ее базовыми принципами, все остальное для профессионала становится простым. Здесь представлен лишь общий обзор ввода и вывода. Подробное описание находится в главах 19 и 20.

Потоки

Программы Java создают потоки ввода-вывода. *Поток* (stream) — это абстракция, которая либо порождает, либо принимает информацию. Поток связан с физическим устройством при помощи системы ввода-вывода Java. Все потоки ведут себя на

один манер, даже несмотря на то, что реальные физические устройства, к которым они подключены, отличаются друг от друга. Таким образом, одни и те же классы и методы ввода-вывода применимы к устройствам разного типа. Это означает, что абстракция входного потока может охватить разные типы ввода: из дискового файла, клавиатуры или сетевого сокета. Аналогично выходной поток может ссылаться на консоль, дисковый файл или сетевое подключение. Потоки — это ясный способ обращения с вводом-выводом без необходимости для вашего кода разбираться в различиях, например, между клавиатурой и сетью. Язык Java реализует потоки внутри иерархии классов, определенных в пакете `java.io`.

На заметку! В дополнение к потоковому вводу-выводу, определенному в пакете `java.io`, язык Java предоставляет также буферы и канальный ввод-вывод, определенные в пакете `java.nio` и его вложенных пакетах. Они описаны в главе 20.

Байтовые и символьные потоки

Java определяет два типа потоков: байтовые и символьные. *Байтовые потоки* предоставляют удобные средства для управления вводом и выводом байтов. Эти потоки используются, например, при чтении и записи бинарных данных. *Символьные потоки* предлагают удобные средства управления вводом и выводом символов. Они используют кодировку Unicode и, таким образом, могут быть интернационализированы. Кроме того, в некоторых случаях символьные потоки более эффективны, чем байтовые.

Исходная версия Java (Java 1.0) не включала символьных потоков, и потому весь ввод-вывод был ориентирован на код виртуальной машины. Символьные потоки были добавлены в Java 1.1, и при этом некоторые ориентированные на код виртуальной машины классы и методы устарели. Хотя старый код, в котором не используются символьные потоки, встречается все реже, он все еще время от времени применяется. Как правило, старый код должен быть, по возможности, обновлен, чтобы воспользоваться преимуществами символьных потоков.

Еще один момент: на самом низком уровне весь ввод-вывод по-прежнему ориентирован на код виртуальной машины. Символьные потоки просто предлагают удобные и эффективные средства управления символами.

Обзор потоков, ориентированных на код виртуальной машины и символы, представлен в следующих разделах.

Классы байтовых потоков

Байтовые потоки определены в двух иерархиях классов. На вершине находятся абстрактные классы `InputStream` и `OutputStream`. Каждый из этих абстрактных классов имеет несколько реальных подклассов, которые контролируют различия между разными устройствами, такими как дисковые файлы, сетевые подключения и даже буферы памяти. Классы байтовых потоков из пакета `java.io` перечислены в табл. 13.1. Некоторые из этих классов описываются ниже, а другие — в части II. Помните, что для использования потоковых классов необходимо импортировать пакет `java.io`.

Таблица 13.1. Классы байтовых потоков из пакета `java.io`

Потоковый класс	Назначение
`BufferedInputStream`	Буферизированный входной поток
`BufferedOutputStream`	Буферизированный выходной поток

Окончание табл. 13.1

Потоковый класс	Назначение
`ByteArrayInputStream`	Входной поток, читающий из массива байт
`ByteArrayOutputStream`	Выходной поток, записывающий в массив байт
`DataInputStream`	Входной поток, включающий методы для чтения стандартных типов данных Java
`DataOutputStream`	Выходной поток, включающий методы для записи стандартных типов данных Java
`FileInputStream`	Входной поток, читающий из файла
`FileOutputStream`	Выходной поток, записывающий в файл
`FilterInputStream`	Реализация класса `InputStream`
`FilterOutputStream`	Реализация класса `OutputStream`
`InputStream`	Абстрактный класс, описывающий поток ввода
`ObjectInputStream`	Входной поток для объектов
`ObjectOutputStream`	Выходной поток для объектов
`OutputStream`	Абстрактный класс, описывающий поток вывода
`PipedInputStream`	Входной канал (например, межпрограммный)
`PipedOutputStream`	Выходной канал
`PrintStream`	Выходной поток, включающий методы `print()` и `println()`
`PushbackInputStream`	Входной поток, поддерживающий однобайтовый возврат во входной поток
`SequenceInputStream`	Входной поток, представляющий собой комбинацию двух и более входных потоков, которые читаются совместно, — один после другого

Абстрактные классы `InputStream` и `OutputStream` определяют несколько ключевых методов, которые реализуют другие потоковые классы. Два наиболее важных — это методы `read()` и `write()`, которые, соответственно, читают и пишут байты данных. Оба метода объявлены как абстрактные внутри классов `InputStream` и `OutputStream`. В наследующих классах они переопределяются.

Классы символьных потоков

Символьные потоки также определены в двух иерархиях классов. На их вершине находятся два абстрактных класса: `Reader` и `Writer`. Эти абстрактные классы управляют потоками символов Unicode. В языке Java предусмотрено несколько конкретных подклассов для каждого из них. Классы символьных потоков перечислены в табл. 13.2.

Таблица 13.2. Классы символьных потоков из пакета `java.io`

Потоковый класс	Назначение
`BufferedReader`	Буферизированный входной символьный поток
`BufferedWriter`	Буферизированный выходной символьный поток
`CharArrayReader`	Входной поток, который читает из символьного массива

Окончание табл. 13.2

Потоковый класс	Назначение
`CharArrayWriter`	Выходной поток, который пишет в символьный массив
`FileReader`	Входной поток, читающий файл
`FileWriter`	Выходной поток, пишущий в файл
`FilterReader`	Фильтрующий читатель
`FilterWriter`	Фильтрующий писатель
`InputStreamReader`	Входной поток, транслирующий байты в символы
`LineNumberReader`	Входной поток, подсчитывающий строки
`OutputStreamWriter`	Выходной поток, транслирующий байты в символы
`PipedReader`	Входной канал
`PipedWriter`	Выходной канал
`PrintWriter`	Выходной поток, включающий методы `print()` и `println()`
`PushbackReader`	Входной поток, позволяющий возвращать символы обратно в поток
`Reader`	Абстрактный класс, описывающий символьный ввод
`StringReader`	Входной поток, читающий из строки
`StringWriter`	Выходной поток, пишущий в строку
`Writer`	Абстрактный класс, описывающий символьный вывод

Абстрактные классы `Reader` и `Writer` определяют несколько ключевых методов, которые реализуют другие потоковые классы. Два наиболее важных — это методы `read()` и `write()`, которые, соответственно читают и пишут символьные данные. Эти методы переопределяются в производных потоковых классах.

Предопределенные потоки

Как вы знаете, все программы Java автоматически импортируют пакет `java.lang`. В этом пакете определен класс `System`, инкапсулирующий некоторые аспекты среды времени выполнения. Например, используя некоторые из его методов, можно получить текущее время и настройки различных параметров, ассоциированных с системой. Класс `System` также содержит три предопределенные потоковые переменные-члена: `in`, `out` и `err`. Эти переменные объявлены в классе `System` как `public`, `static` и `final`. Это значит, что они могут быть использованы любой другой частью вашей программы без обращения к специфическому объекту класса `System`.

Переменная `System.out` ссылается на стандартный выходной поток. По умолчанию это консоль. Переменная `System.in` ссылается на стандартный входной поток, который также по умолчанию является консолью. Переменная `System.err` ссылается на стандартный поток ошибок, который также по умолчанию связан с консолью. Однако эти потоки могут быть перенаправлены на любое совместимое устройство ввода-вывода.

Переменная `System.in` — это объект класса `InputStream`, а переменные `System.out` и `System.err` — объекты класса `PrintStream`. Это байтовые потоки, хотя обычно они используются для чтения и записи символов с консоли и

на консоль. Как вы увидите, при необходимости их можно поместить в оболочки символьных потоков.

В примерах, приведенных в предыдущих главах, использовался поток `System.out`. Вы можете почти таким же образом применять поток `System.err`. Как будет показано в следующем разделе, использование потока `System.in` немного сложнее.

Чтение консольного ввода

В языке Java 1.0 единственным способом выполнения консольного ввода было использование байтового потока. Сегодня применение байтового потока для чтения консольного ввода по-прежнему возможно. Однако для коммерческих приложений предпочтительный метод чтения консольного ввода – это использовать символьный поток. Это значительно упрощает возможности интернационализации и поддержки разрабатываемых программ.

В языке Java консольный ввод осуществляется чтением потока `System.in`. Чтобы получить символьный поток, присоединенный к консоли, следует поместить поток `System.in` в оболочку объекта класса `BufferedReader`. Объект класса `BufferedReader` поддерживает буферизованный входной поток. Его наиболее часто используемый конструктор выглядит так.

```
BufferedReader(Reader считывательВвода)
```

Здесь *считывательВвода* – это поток, который связывается с создаваемым экземпляром класса `BufferedReader`. Класс `Reader` – абстрактный класс. Одним из его конкретных наследников является класс `InputStreamReader`, который преобразует байты в символы. Для получения объекта класса `InputStreamReader`, который присоединен к потоку `System.in`, служит следующий конструктор.

```
InputStreamReader(InputStream потокВвода)
```

Поскольку переменная `System.in` ссылается на объект класса `InputStream`, она должна быть использована как параметр *потокВвода*. Собрав все вместе, получим следующую строку кода, которая создает экземпляр класса `BufferedReader`, связанный с клавиатурой.

```
BufferedReader br = new BufferedReader(new
                        InputStreamReader(System.in));
```

После выполнения этого оператора объект `br` представляет собой символьный поток, подключенный к консоли через поток `System.in`.

Чтение символов

Для чтения символа из объекта класса `BufferedReader` применяется метод `read()`. Ниже показана версия метода `read()`, которая будет использоваться.

```
int read() throws IOException
```

Каждый раз, когда вызывается метод `read()`, он читает символ из входного потока и возвращает его как целочисленное значение. При достижении конца потока возвращается значение `-1`. Как видите, метод может передать исключение `IOException`.

В следующей программе демонстрируется применение метода `read()` для чтения символов с консоли до тех пор, пока пользователь не введет `"q"`. Обратите внимание на то, что любые исключения ввода-вывода, которые могут быть созданы, просто передаются в метод `main()`. Такой подход распространен при чтении

с консоли в простых примерах программ, таких как показаны в этой книге, но в более сложных приложениях вы можете обработать исключения явно.

```
// Использование BufferedReader для чтения символов с консоли.
import java.io.*;

class BRRead {
    public static void main(String args[]) throws IOException
    {
        char c;
        BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));
        System.out.println("Вводите символы, 'q' - для выхода.");
        // читать символы
        do {
            c = (char) br.read();
            System.out.println(c);
        } while(c != 'q');
    }
}
```

Ниже показан пример запуска этой программы.

```
Вводите символы, 'q' - для выхода.
123abcq
1
2
3
a
b
c
q
```

Этот вывод может выглядеть немного не так, как вы ожидали, потому что поток `System.in` является строчно-буферизованным по умолчанию. Это значит, что никакого ввода в действительности программе не передается до тех пор, пока не будет нажата клавиша <Enter>. Как можно предположить, это делает метод `read()` лишь отчасти применимым для интерактивного консольного ввода.

Чтение строк

Чтобы прочесть строку с клавиатуры, используйте версию метода `readLine()`, который является членом класса `BufferedReader`. Его общая форма такова.

```
String readLine() throws IOException
```

Как видите, он возвращает объект класса `String`.

Следующая программа демонстрирует объект класса `BufferedReader` и метод `readLine()`. Программа читает и отображает строки текста до тех пор, пока вы не введете слово "стоп".

```
// Чтение строк с консоли с применением BufferedReader.
import java.io.*;

class BRReadLines {
    public static void main(String args[]) throws IOException
    {
        // Создать BufferedReader с использованием System.in
        BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));
        String str;
```

```
        System.out.println("Вводите строки текста.");
        System.out.println("Введите 'стоп' для завершения.");
        do {
            str = br.readLine();
            System.out.println(str);
        } while(!str.equals("стоп"));
    }
}
```

В следующем примере создается крошечный текстовый редактор. В коде создается массив объектов класса `String`, а затем читаются строки текста с сохранением каждой строки в виде элемента массива. Чтение производится до 100 строк или до того, как будет введено слово "стоп". Для чтения с консоли используется объект класса `BufferedReader`.

```
// Крошечный редактор.
import java.io.*;

class TinyEdit {
    public static void main(String args[]) throws IOException
    {
        // Создать BufferedReader, используя System.in
        BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));
        String str[] = new String[100];
        System.out.println("Вводите строки текста.");
        System.out.println("Введите 'стоп' для завершения.");
        for(int i=0; i<100; i++) {
            str[i] = br.readLine();
            if(str[i].equals("стоп")) break;
        }
        System.out.println("\nВот ваш файл:");
        // отобразить строки
        for(int i=0; i<100; i++) {
            if(str[i].equals("стоп")) break;
            System.out.println(str[i]);
        }
    }
}
```

Ниже показан пример запуска этой программы.

```
Вводите строки текста.
Введите 'стоп' для завершения.
Это строка один.
Это строка два.
Java делает работу со строками простой.
Просто создайте объект String.
стоп
Вот ваш файл:
Это строка один.
Это строка два.
Java делает работу со строками простой.
Просто создайте объект String.
```

Запись консольного вывода

Консольный вывод проще всего осуществлять с помощью описанных ранее методов `print()` и `println()`, которые используются в большинстве примеров

этой книги. Эти методы определены в классе PrintStream (который является типом переменной System.out). Даже несмотря на то, что поток System.out — байтовый, применение его для вывода в простых программах вполне оправдано. Тем не менее в следующем разделе описана его символьная альтернатива.

Поскольку класс PrintStream описывает выходной поток и происходит от класса OutputStream, он также реализует низкоуровневый метод write(). То есть метод write() может применяться для записи на консоль. Простейшая форма метода write(), определенного в классе PrintStream, показана ниже.

```
void write(int значениебайта)
```

Этот метод запишет байт, переданный в параметре *значениебайта*. Хотя параметр *значениебайта* объявлен как целочисленный, записываются только 8 его младших бит. Вот короткий пример, использующий метод write() для вывода на экран буквы "A" с последующим переводом строки.

```
// Демонстрация System.out.write().
class WriteDemo {
    public static void main(String args[]) {
        int b;

        b = 'A';
        System.out.write(b);
        System.out.write('\n');
    }
}
```

Вам не часто придется использовать метод write() для вывода на консоль (хотя в некоторых ситуациях это и удобно), поскольку значительно проще применять для этого методы print() и println().

Класс PrintWriter

Хотя применение потока System.out для вывода на консоль допустимо, он, вероятно, лучше подходит для отладки или для примеров программ вроде тех, что приводятся в настоящей книге. Для реальных программ рекомендуемым способом записи на консоль является поток класса PrintWriter. Класс PrintWriter — это один из символьных классов. Применение такого класса для консольного вывода упрощает интернационализацию ваших программ.

Класс PrintWriter определяет несколько конструкторов. Один из тех, которые мы будем использовать, показан ниже.

```
PrintWriter(OutputStream потокВывода, boolean сбросПриНовойСтроке)
```

Здесь *потокВывода* — объект класса OutputStream, а *сбросПриНовойСтроке* управляет тем, будет ли Java сбрасывать буфер в выходной поток каждый раз при вызове метода println(). Если значение *сбросПриНовойСтроке* равно true, то происходит автоматический сброс буфера, если же false, то нет.

Класс PrintWriter поддерживает методы print() и println(). То есть вы можете использовать эти методы таким же способом, как они применяются в потоке System.out. Если аргумент не простого типа, то объект класса PrintWriter вызывает метод toString(), а затем выводит результат.

Чтобы писать на консоль с помощью объекта класса PrintWriter, укажите поток System.out в качестве выходного потока и сбрасывайте поток после каждого символа новой строки. Например, следующая строка кода создает объект класса PrintWriter, который подключен к консольному выводу.

```
PrintWriter pw = new PrintWriter(System.out, true);
```

Показанное ниже приложение иллюстрирует применение класса `PrintWriter` для управления консольным выводом.

```
// Демонстрация PrintWriter
import java.io.*;

public class PrintWriterDemo {
    public static void main(String args[]) {
        PrintWriter pw = new PrintWriter(System.out, true);

        pw.println("Это строка");
        int i = -7;
        pw.println(i);
        double d = 4.5e-7;
        pw.println(d);
    }
}
```

Вывод этой программы будет выглядеть следующим образом.

```
Это строка
-7
4.5E-7
```

Помните, что нет ничего неправильного в применении потока `System.out` для простого текстового вывода на консоль, когда вы изучаете язык Java или занимаетесь отладкой своих программ. Однако класс `PrintWriter` обеспечивает возможность простой интернационализации для реальных программ. Поскольку никаких выгод от использования класса `PrintWriter` в простых программах нет, мы продолжим пользоваться потоком `System.out` для вывода на консоль.

Чтение и запись файлов

Язык Java предоставляет множество классов и методов, которые позволяют вам читать и записывать файлы. Прежде чем мы начнем, следует сказать, что тема ввода-вывода в файл весьма обширна; подробно она исследуется в части II. Задача этого раздела в том, чтобы познакомить вас с основными технологиями чтения из файла и записи в него. Хотя используются байтовые потоки, эти технологии можно адаптировать к символьным потокам.

Два наиболее часто используемых потоковых класса — это классы `FileInputStream` и `FileOutputStream`, которые создают байтовые потоки, связанные с файлами. Чтобы открыть файл, вы просто создаете объект одного из этих классов, указав имя файла в качестве аргумента конструктора. Хотя оба класса имеют и дополнительные конструкторы, мы будем использовать только следующие.

```
FileInputStream(String имяФайла) throws FileNotFoundException
FileOutputStream(String имяФайла) throws FileNotFoundException
```

Здесь *имяФайла* — имя файла, который вы хотите открыть. Если при создании входного потока файл не существует, передается исключение `FileNotFoundException`. Для выходных потоков, если файл не может быть открыт или создан, также передается исключение `FileNotFoundException`. Класс исключения `FileNotFoundException` происходит от класса `IOException`. Когда выходной файл открыт, любой ранее существовавший файл с тем же именем уничтожается.

На заметку! В ситуациях, где присутствует менеджер безопасности, некоторые файловые классы, включая классы `FileInputStream` и `FileOutputStream`, передают исключение `SecurityException`, если при попытке открыть файл произойдет нарушение безопасности. По умолчанию приложения, запущенные при помощи команды `java`, не используют менеджер безопасности. Поэтому примеры ввода-вывода в этой книге не обязаны отслеживать возможность передачи исключения `SecurityException`. Однако другие типы приложений (такие, как аплеты) будут использовать менеджер безопасности, и их файловый ввод-вывод вполне может создать исключение `SecurityException`. В таком случае вы будете должны соответственно обработать и это исключение.

Когда вы завершаете работу с файлом, его необходимо закрыть. Для этого используется метод `close()`, реализованный в классах `FileInputStream` и `FileOutputStream`, как показано ниже.

```
void close() throws IOException
```

Закрытие файла высвобождает выделенные для него системные ресурсы, позволяя использовать их для других файлов. Неудача закрытия файла может привести к "утечке памяти", поскольку неиспользуемые ресурсы останутся зарезервированы.

На заметку! Начиная с JDK 7 метод `close()` определяется интерфейсом `AutoCloseable` в пакете `java.lang`. Интерфейс `AutoCloseable` унаследован интерфейсом `Closeable` в пакете `java.io`. Оба интерфейса реализуются потоковыми классами, включая классы `FileInputStream` и `FileOutputStream`.

Следует заметить, что существует два основных подхода, которые вы можете использовать для закрытия файла, когда он больше не нужен. Первый – традиционный подход, при котором метод `close()` вызывается явно, когда файл больше не нужен. Этот подход используется всеми версиями Java до JDK 7 и потому находится во всем существующем коде. Второй подразумевает использование нового оператора `try`-*с-ресурсами*, добавленного в JDK 7. Он автоматически закрывает файл, когда он больше не нужен. При этом подходе нет никаких явных вызовов метода `close()`. Поскольку существуют миллионы строк кода, написанного до JDK 7, которые все еще используются и поддерживаются, важно знать и понимать традиционный подход. Поэтому начнем с него. Новый автоматизированный подход описывается в следующем разделе.

Чтобы читать файл, вы можете применять версию метода `read()`, которая определена в классе `FileInputStream`. Та, что мы будем использовать, выглядит так.

```
int read() throws IOException
```

Всякий раз, когда вызывается этот метод, он читает единственный байт из файла и возвращает его как целое число. Метод `read()` возвращает значение `-1`, когда достигнут конец файла. Метод может передать исключение `IOException`.

В следующей программе метод `read()` используется для ввода и отображения содержимого файла с текстом ASCII. Имя файла указано в аргументе командной строки.

```
/* Отображение текстового файла.
   Чтобы использовать эту программу, укажите
   имя файла, который хотите просмотреть.
   Например, чтобы просмотреть файл TEST.TXT,
   используйте следующую командную строку:

   java ShowFile TEST.TXT
*/

import java.io.*;
```

```
class ShowFile {
    public static void main(String args[])
    {
        int i;
        FileInputStream fin;

        // Сначала убедиться, что имя файла указано.
        if(args.length != 1) {
            System.out.println("Использование: ShowFile Файл");
            return;
        }

        // Попытка открыть файл.
        try {
            fin = new FileInputStream(args[0]);
        } catch(FileNotFoundException e) {
            System.out.println("Не могу открыть файл");
            return;
        }

        // Теперь файл открыт и готов к чтению.
        // Следующий код читает символы, 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 чтения файла");
        }

        // Закрыть файл.
        try {
            fin.close();
        } catch(IOException e) {
            System.out.println("Ошибка закрытия файла");
        }
    }

}
```

В программе обратите внимание на блок `try/catch`, обрабатывающий ошибки ввода-вывода, которые могут произойти. Каждая операция ввода-вывода проверяется на исключения, и если исключение происходит, оно обрабатывается. В простых программах или коде примеров вполне обычна передача исключений ввода-вывода за пределы функции `main()`, как это делалось в предыдущих консольных примерах. Кроме того, в реальном коде иногда полезно позволить исключению распространиться на вызывающую подпрограмму, чтобы уведомить ее о неудаче операции ввода-вывода. Однако большинство примеров файлового ввода-вывода в этой книге обрабатывает все исключения ввода-вывода явно, для демонстрации.

Хотя приведенный пример закрывает файловый поток после чтения файла, существует вариант, который зачастую полезен. Он подразумевает вызов метода `close()` в пределах блока `finally`. При таком подходе все методы, которые получают доступ к файлу, содержатся в пределах блока `try`, а блок `finally` используется для закрытия файла. Таким образом, независимо от того как закончится блок `try`, файл будет закрыт. С учетом приведенного примера, вот как может быть переделан блок `try`, 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 чтения файла");
} finally {
    // Закрыть файл при выходе из блока try.
    try {
        fin.close();
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла");
    }
}
```

Хотя в данном случае это не проблема, одним из преимуществ такого подхода является то, что если обращающийся к файлу код прекращает работу из-за каких-либо не связанных с вводом-выводом исключений, файл все равно будет закрыт блоком `finally`.

Иногда проще заключить все части программы, которые открывают файл и получают доступ к его содержимому, в один блок `try` (вместо того чтобы разделять его на два), а затем использовать блок `finally`, чтобы закрыть файл. Вот, например, другой способ написать программу `ShowFile`.

```
/* Отображение текстового файла.
   Чтобы использовать эту программу, укажите
   имя файла, который хотите просмотреть.
   Например, чтобы просмотреть файл TEST.TXT,
   используйте следующую командную строку:

   java ShowFile TEST.TXT

   Этот вариант заключает код, который открывает
   и получает доступ к файлу, в один блок try.
   Файл закрывает блок finally.
*/

import java.io.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;
        FileInputStream fin = null;

        // Сначала убедиться, что имя файла указано.
        if(args.length != 1) {
            System.out.println("Использование: ShowFile Файл");
            return;
        }

        // Следующий код открывает файл, читает символы, пока не
        // встретится EOF, а затем закрывает файл в блоке 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("Произошла ошибка I/O");
        } finally {
            // Закрыть файл в любом случае.
            try {
                if(fin != null) fin.close();
            } catch(IOException e) {
                System.out.println("Ошибка закрытия файла");
            }
        }
    }
}
```

Обратите внимание на то, что при этом подходе объект `fin` инициализируется значением `null`. Затем, в блоке `finally`, файл закрывается, только если объект `fin` не содержит значение `null`. Это работает потому, что объект `fin` не будет содержать значение `null`, только если файл был успешно открыт. Таким образом, метод `close()` не вызывается, если при открытии файла происходит исключение.

Последовательность операторов `try/catch` в приведенном выше примере можно сделать более компактной. Поскольку класс исключения `FileNotFoundException` происходит от класса `IOException`, его не обязательно обрабатывать отдельно. Вот, например, переделанная последовательность операторов, устраняющая обработчик исключения `FileNotFoundException`. В данном случае отображается стандартное сообщение исключения, 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 I/O: " + e);
} finally {
    // Закрыть файл в любом случае.
    try {
        if(fin != null) fin.close();
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла");
    }
}
```

При этом подходе любая ошибка, включая ошибку открытия файла, обрабатывается одним оператором `catch`. Благодаря компактности, этот подход используется в большинстве примеров ввода-вывода данной книги. Однако не забывайте, что этот подход не является подходящим в случаях, когда вы хотите по-разному реагировать на разные неудачи открытия файла, например, когда пользователь может неправильно ввести имя файла. В такой ситуации вы могли бы, например, запросить правильное имя, прежде чем переходить к блоку `try`, который обращается к файлу.

Для записи в файл вы будете использовать метод `write()`, определенный в классе `FileOutputStream`. Его простейшая форма выглядит так.

```
void write(int значениебайта) throws IOException
```

Этот метод пишет в файл байт, переданный параметром *значениебайта*. Хотя параметр *значениебайта* объявлен как целочисленный, в файл записываются только его младшие восемь бит. Если при записи произойдет ошибка, передается исключение `IOException`. В следующем примере метод `write()` используется для копирования файла.

```
/* Копирование файла.
   Для использования этой программы укажите
   имена исходного и целевого файлов.
   Например, чтобы скопировать файл FIRST.TXT в файл
   SECOND.TXT, используйте 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;

        // Сначала убедиться, что указаны имена обоих файлов.
        if(args.length != 2) {
            System.out.println("Использование: CopyFile из в");
            return;
        }

        // Копирование файла.
        try {
            // Попытка 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 I/O: " + e);
        } finally {
            try {
                if(fin != null) fin.close();
            } catch(IOException e2) {
                System.out.println("Ошибка закрытия файла ввода");
            }
            try {
                if(fout != null) fout.close();
            } catch(IOException e2) {
                System.out.println("Ошибка закрытия файла вывода");
            }
        }
    }
}
```

Обратите внимание на то, что в программе при закрытии файлов используются два отдельных блока `try`. Это гарантирует, что оба файла будут закрыты, даже

если вызов метода `fin.close()` передает исключение. Обратите также внимание на то, что все потенциальные ошибки ввода-вывода обрабатываются в двух приведенных выше программах при помощи исключений. В некоторых языках для сообщения о файловых ошибках используются коды ошибок. Это не только делает управление файлами понятнее, но и позволяет Java просто отличать условие достижения конца файла от файловых ошибок во время ввода. В языке C/C++ многие функции ввода возвращают одно и то же значение, когда происходит ошибка и когда достигается конец файла. (То есть в языке C/C++ условие EOF часто накладывается на то же значение, что и ошибка ввода.) Обычно это означает, что программист обязан включать дополнительные операторы для определения того, какое событие на самом деле произошло. В языке Java ошибки ввода передаются программе в виде исключений, а не через значение, возвращаемое методом `read()`. Другими словами, когда метод `read()` возвращает значение `-1`, это значит только одно: достигнут конец файла.

Автоматическое закрытие файла

В приведенном выше разделе примеры программ осуществляли явный вызов метода `close()`, чтобы закрыть файл, как только он окажется ненужным. Как уже упоминалось, это способ закрытия файлов до использования в Java комплекта JDK 7. Хотя этот подход все еще допустим и применим, в JDK 7 добавлена новая возможность, предлагающая иной способ управления ресурсами, такой как файловые потоки при автоматическом завершении процесса. Это средство иногда называется *автоматическим управлением ресурсами* (automatic resource management – ARM) и основано на усовершенствованной версии оператора `try`. Основное преимущество автоматического управления ресурсами заключается в предотвращении ситуаций, когда файл (или другой ресурс) по неосторожности не освобождается после того, как он больше не нужен. Как уже упоминалось, незакрытые файлы, о которых забыли, могут привести к утечке памяти и многим другим проблемам.

Автоматическое управление ресурсами основано на усовершенствованной форме оператора `try`. Вот его общая форма.

```
try (спецификация_ресурса) {
    // использование ресурса
}
```

Здесь *спецификация_ресурса* – это оператор, который объявляет и инициализирует ресурс, такой как файловый поток. Он состоит из объявления переменной, в котором переменная инициализируется ссылкой на используемый объект. По завершении блока `try` ресурс автоматически освобождается. В случае файла это означает, что файл автоматически закрывается. (Таким образом, нет никакой необходимости вызывать метод `close()` явно.) Конечно, эта форма оператора `try` может также включать директивы `finally` и `catch`. Эта новая форма оператора `try` называется оператор `try`-*с-ресурсами* (`try`-with-resources).

Оператор `try`-*с-ресурсами* применяется только с теми ресурсами, которые реализуют интерфейс `AutoCloseable`, определенный в пакете `java.lang`. Этот интерфейс определяет метод `close()`. Интерфейс `AutoCloseable` унаследован интерфейсом `Closeable` в пакете `java.io`. Оба интерфейса реализуются потоковыми классами. Таким образом, оператор `try`-*с-ресурсами* может быть использован при работе с потоками, включая файловые потоки.

В качестве первого примера автоматического закрытия файла рассмотрим переделанную версию программы `ShowFile`.

```
/* Эта версия программы ShowFile использует оператор try-с-ресурсами,
   чтобы автоматически закрыть файл.

   Примечание: Этот код требует JDK 7.
*/

import java.io.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;

        // Сначала убедиться, что имя файла указано.
        if(args.length != 1) {
            System.out.println("Использование: ShowFile filename");
            return;
        }

        // Следующий код использует оператор try-с-ресурсами, чтобы
        // открыть файл, а затем автоматически закрыть его, когда блок
        // try завершится.
        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 I/O");
        }
    }
}
```

Обратите в коде особое внимание на то, как файл открывается в пределах оператора `try`.

```
try(FileInputStream fin = new FileInputStream(args[0])) {
```

Обратите внимание на то, как часть спецификации ресурса `try` объявляет экземпляр класса `FileInputStream` по имени `fin`, которому затем присваивается ссылка на файл, открытый его конструктором. Таким образом, в данной версии программы переменная `fin` является локальной по отношению к блоку `try`, в начале которого она создается. При завершении блока `try` поток, связанный с переменной `fin`, автоматически закрывается неявным вызовом метода `close()`. Вы не должны вызывать метод `close()` явно, а это значит, что вы не можете забыть закрыть файл. Это главное преимущество использования оператора *try-с-ресурсами*.

Важно понять, что ресурс, объявленный в операторе `try`, является неявно финальным. Это значит, что вы не можете повторно присвоить ресурс после того, как он был создан. Кроме того, область видимости ресурса ограничивается оператором *try-с-ресурсами*.

Вы можете управлять несколькими ресурсами в пределах одного оператора `try`. Для этого просто отделите каждую спецификацию ресурса точкой с запятой. Следующая программа демонстрирует пример. Здесь программа `CopyFile` пере-

делана так, чтобы использовать один оператор *try-с-ресурсами* для работы и с переменными fin и fout.

```
/* Версия программы CopyFile, использующая оператор try-с-ресурсами.
   Она демонстрирует управление двумя ресурсами (в данном случае
   файлами) в одном операторе try.
*/

import java.io.*;

class CopyFile {
    public static void main(String args[]) throws IOException
    {
        int i;

        // Сначала убедиться, что заданы оба файла.
        if(args.length != 2) {
            System.out.println("Использование: CopyFile из в");
            return;
        }

        // Открыть и управлять двумя файлами в операторе 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 I/O: " + e);
        }
    }
}
```

Обратите внимание на то, как файлы ввода и вывода открываются в пределах блока try.

```
try (FileInputStream fin = new FileInputStream(args[0]);
     FileOutputStream fout = new FileOutputStream(args[1]))
{
    // ...
```

После завершения этого блока try будут закрыты как fin, так и fout. Если сравнить эту версию программы с предыдущей, то можно заметить, что она намного короче. Возможность упростить исходный код является дополнительным преимуществом автоматического управления ресурсами.

У оператора *try-с-ресурсами* есть еще один аспект, который стоит упомянуть. Вообще, когда выполняется блок try, есть вероятность того, что исключение в блоке try приведет к другому исключению, которое произойдет тогда, когда ресурс закрывается в директиве finally. В случае "обычного" оператора try, первоначальное исключение теряется, будучи вытесненным вторым исключением. Но при использовании оператора *try-с-ресурсами* второе исключение *подавляется* (suppressed). Однако оно не теряется. Вместо этого оно добавляется в список подавленных исключений, связанных с первым исключением. Доступ к списку подавленных исключений может быть получен при помощи метода getSuppressed(), определенного в классе Throwable.

Благодаря преимуществам оператора *try-с-ресурсами*, он будет использован во многих, но не во всех примерах программ данной книги. В некоторых примерах все еще будет использоваться традиционный подход закрытия ресурсов. Для этого есть несколько причин. Во-первых, существуют миллионы строк широко распространенного и используемого кода, который полагается на традиционный подход. Важно то, что все программисты Java хорошо знакомы с традиционным подходом. Во-вторых, не все разрабатываемые проекты немедленно перейдут на новую версию JDK. Некоторые программисты, вероятно, какое-то время продолжат работать в среде, предшествующей JDK 7. В таких ситуациях улучшенная форма оператора `try` недоступна. И наконец, могут быть случаи, в которых явное закрытие ресурса лучше, чем автоматический подход. По этим причинам в некоторых примерах книги будет продолжено использование традиционного подхода путем явного вызова метода `close()`. В дополнение к иллюстрированию традиционной методики, эти примеры могут быть также откомпилированы и запущены всеми читателями на всех системах.

На заметку! В некоторых примерах книги используется традиционный подход закрытия файлов как средство демонстрации данной технологии, которая широко используется в существующем коде. Однако для нового кода желательно использовать новый автоматизированный подход, поддерживаемый только что описанным оператором *try-с-ресурсами*.

Основы организации аплетов

Все предыдущие примеры программ были консольными приложениями Java. Однако приложения этого типа — только один класс программ Java. Другой тип программ Java — аплеты. Как упоминалось в главе 1, *аплет* – это маленькое приложение, которое находится на интернет-сервере, транспортируется по Интернету, автоматически инсталлируется и запускается как часть веб-документа. После того как аплет появляется у клиента, он получает ограниченный доступ к ресурсам так, что обеспечивает сложный графический пользовательский интерфейс и выполняет сложные вычисления, не подвергая клиента риску вирусной атаки или повреждения целостности его данных.

Многие вопросы создания и применения аплетов будут рассмотрены в части II, где представлен пакет `applet`. Однако основы, имеющие отношение к созданию аплетов, рассмотрим прямо сейчас, поскольку аплеты имеют структуру, отличную от программ, с которыми мы имели дело до сих пор в этой книге. Как вы увидите, аплеты отличаются от приложений по нескольким ключевым признакам.

Начнем с простейшего аплета.

```
import java.awt.*;
import java.applet.*;

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Простейший аплет", 20, 20);
    }
}
```

Этот аплет начинается с двух операторов `import`. Первый импортирует классы библиотеки Abstract Window Toolkit (AWT). Аплеты взаимодействуют с пользователем (непосредственно или опосредованно) через библиотеку AWT, а не через классы консольного ввода-вывода. Как вы можете предположить, библиотека AWT значительно больше и сложнее, и полное обсуждение ее возможностей занимает

несколько глав в части II книги. К счастью, этот простой аплет очень ограниченно использует библиотеку AWT. (Аплеты также могут использовать библиотеку Swing для предоставления графического пользовательского интерфейса, но этот подход рассматривается далее в книге.) Второй оператор `import` импортирует пакет `applet`, в котором находится класс `Applet`. Каждый аплет, который вы создадите, должен быть подклассом (прямо или косвенно) класса `Applet`.

Следующая строка в программе объявляет класс `SimpleApplet`. Этот класс должен быть объявлен открытым (`public`), чтобы быть доступным коду вне нашей программы.

Внутри класса `SimpleApplet` объявлен метод `paint()`. Этот метод определен библиотекой AWT и должен быть переопределен аплетом. Метод `paint()` вызывается всякий раз, когда аплет должен перерисовать свой вывод. Эта ситуация может возникнуть по нескольким причинам. Например, окно, в котором запущен аплет, может быть перекрыто другим окном, а затем вновь открыто. Или же окно аплета может быть минимизировано, а затем восстановлено. Метод `paint()` также вызывается, когда аплет начинает выполнение. Независимо от причины, всякий раз, когда аплет должен перерисовать свое содержимое, вызывается метод `paint()`. Метод `paint()` принимает один параметр типа `Graphics`. Этот параметр содержит графический контекст, который описывает графическую среду, в которой работает аплет. Этот контекст используется всякий раз, когда запрашивается его вывод.

Внутри метода `paint()` вызывается метод `drawString()`, являющийся методом класса `Graphics`. Этот метод выводит строку в позиции, заданной координатами X,Y. Он имеет следующую общую форму.

```
void drawString(String сообщение, int x, int y)
```

Здесь *сообщение* — это строка, которая должна быть выведена начиная с позиции *x*, *y*. В окне Java верхний левый угол имеет координаты 0,0. Вызов метода `drawString()` в аплете отображает строку "Простейший аплет" начиная с позиции 20,20.

Обратите внимание на то, что аплет не имеет метода `main()`. В отличие от программ Java, аплет не начинает выполнение с метода `main()`. Фактически большинство аплетов даже не имеет этого метода. Вместо этого аплет начинает выполнение, когда имя его класса передается средству просмотра аплетов или сетевому браузеру.

После ввода исходного текста аплета `SimpleApplet` его компиляция выполняется так же, как компиляция обычных программ. Однако запуск аплета `SimpleApplet` осуществляется иначе. Фактически есть два способа, которыми можно запустить аплет.

- Выполнение аплета внутри совместимого с Java браузера.
- Использование средства просмотра аплетов, такого как стандартный инструмент `appletviewer`. Он выполняет ваш аплет в окне. Обычно это самый быстрый и простой способ проверки аплета.

Ниже подробно описывается каждый из этих способов.

Один из способов выполнить аплет в веб-браузере — это написать короткий файл HTML, который должен содержать соответствующий дескриптор. В настоящее время Oracle рекомендует использовать дескриптор `APPLET`. (Также может быть использован дескриптор `OBJECT`. Более подробная информация о стратегиях развертывания аплетов приведена в главе 22.) Для использования дескриптора `APPLET` здесь применяется файл HTML, запускающий аплет `SimpleApplet`.

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

Параметры `width` и `height` указывают размеры области отображения, используемой аплетом. (Дескриптор `APPLET` содержит несколько других параметров, которые рассматриваются более подробно в части II книги.) После того как создадите этот файл, следует запустить браузер, а затем загрузить в него этот файл, что вызовет выполнение аплета `SimpleApplet`.

Чтобы выполнить аплет `SimpleApplet` в средстве просмотра аплетов, вы также должны выполнить файл HTML, показанный выше. Например, если предыдущий файл HTML называется `RunApp.html`, то следующая командная строка запустит его на выполнение.

```
C:\>appletviewer RunApp.html
```

Однако существует более удобный метод, который ускорит проверку. Просто включите комментарий в начало исходного файла Java, который указан в дескрипторе `APPLET`. В результате ваш код будет документирован прототипом необходимых конструкций HTML, и вы сможете проверить скомпилированный аплет, запуская средство просмотра аплетов с указанием исходного файла Java. Если вы применяете этот метод, то исходный файл `SimpleApplet` должен выглядеть следующим образом.

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

При таком подходе вы сможете быстро проходить этапы разработки аплета, выполняя перечисленные ниже три шага.

1. Редактирование файла исходного кода Java.
2. Компиляция программы.
3. Запуск средства просмотра аплетов с указанием имени исходного файла. Средство просмотра аплетов обнаружит дескриптор `APPLET` внутри комментария и запустит аплет.

Окно аплета `SimpleApplet`, отображенное средством просмотра аплетов, показано на рис. 13.1.

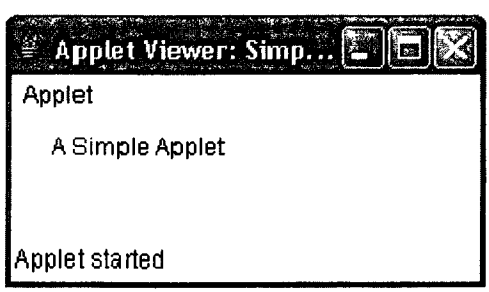


Рис. 13.1. Аплет `SimpleApplet` во время выполнения

Хотя сущность аплетов будет обсуждаться далее в этой книге, здесь мы укажем ключевые моменты, о которых нужно знать сейчас.

- Аплеты должны запускаться под управлением средства просмотра аплетов или совместимого с Java веб-браузера.
- Пользовательский ввод-вывод в аплетах не выполняется с использованием классов ввода-вывода. Вместо этого аплеты применяют интерфейс, предоставляемый средствами библиотек AWT или Swing.

Модификаторы `transient` и `volatile`

Язык Java определяет два интересных модификатора типов: `transient` и `volatile`. Эти модификаторы служат для управления некоторыми специфическими ситуациями.

Когда экземпляр переменной объявлен как `transient`, его значение не должно сохраняться, когда объект сохраняется.

```
class T {
    transient int a; // не будет сохраняться
    int b;           // будет сохраняться
}
```

Здесь, если объект типа `T` записывается в область постоянного хранения, содержимое `a` не должно сохраняться, а содержимое `b` — должно быть сохранено.

Модификатор `volatile` сообщает компилятору, что отмеченная им переменная может быть неожиданно изменена другими частями вашей программы. Одна из таких ситуаций возникает в многопоточных программах. В многопоточных программах иногда два или более потоков имеют совместный доступ к одной и той же переменной. Из соображений эффективности, каждый поток может хранить свою собственную закрытую копию этой переменной. Реальная копия (или *мастер-копия*) переменной обновляется в различные моменты, например при входе в метод `synchronized`. Хотя такой подход работает нормально, все же иногда он недостаточно эффективен. В некоторых случаях все, что действительно происходит, — это то, что мастер-копия переменной всегда отражает ее текущее состояние. Чтобы обеспечить это, просто объявите переменную как `volatile`, что сообщит компилятору о необходимости всегда использовать мастер-копию этой переменной (или же, как минимум, всегда держать закрытые ее копии синхронизированными с мастер-копией и наоборот). Кроме того, доступ к мастер-копии переменной должен осуществляться в том же порядке, как он выполнялся к закрытой копии.

Использование оператора `instanceof`

Иногда может понадобиться узнать тип объекта во время выполнения программы. Например, вы можете иметь один поток выполнения, который генерирует объекты различных типов, и другой поток, который их использует. В этой ситуации для обрабатывающего потока может быть удобно знать тип каждого объекта, который он получает. Другая ситуация, когда знание типа объекта во время выполнения важно, — это когда используется приведение типа. В языке Java неправильное приведение типа вызывает ошибку времени выполнения. Множество неверных приведений типа могут быть перехвачены на этапе компиляции. Однако

приведение типов в пределах иерархии классов может стать причиной ошибок приведения, которые обнаруживаются только во время выполнения. Например, суперкласс по имени A может порождать два подкласса — B и C. Таким образом, приведение объекта класса B к типу A или C к A допустимо, но приведение объекта класса B к типу C (и наоборот) — некорректно. Поскольку объект класса A может ссылаться на объекты и класса B, и класса C, как вы можете узнать во время выполнения, к какому именно типу обращается ссылка перед тем, как осуществить приведение к типу C? Это может быть объект класса A, B или C. Если это объект класса B, то будет передано исключение времени выполнения. Для получения ответа на этот вопрос Java предлагает оператор времени выполнения `instanceof`.

Общая форма оператора `instanceof` такова.

ссылканаобъект `instanceof` *тип*

Здесь *ссылканаобъект* — ссылка на экземпляр класса, а *тип* — тип класса. Если *ссылканаобъект* относится к указанному типу или может быть приведена к нему, то оператор `instanceof` дает в результате `true`. В противном случае результатом будет `false`. То есть оператор `instanceof` — это средство, с помощью которого программа может получить информацию об объекте во время выполнения.

В следующей программе демонстрируется применение оператора `instanceof`.

```
// Демонстрация использования оператора 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 есть экземпляр A");
        if(b instanceof B)
            System.out.println("b есть экземпляр B");
        if(c instanceof C)
            System.out.println("c есть экземпляр C");
        if(c instanceof A)
            System.out.println("c может быть приведен к A");

        if(a instanceof C)
            System.out.println("a может быть приведен к C");

        System.out.println();

        // сравнение типов с порожденными типами
        A ob;
```

```
        ob = d; // Ссылка на d
        System.out.println("ob теперь ссылается на d");
        if(ob instanceof D)
            System.out.println("ob есть экземпляр 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 может быть приведен к D");
        else
            System.out.println("ob не может быть приведен к D");

        if(ob instanceof A)
            System.out.println("ob может быть приведен к A");

        System.out.println();

        // все объекты могут быть приведены к Object
        if(a instanceof Object)
            System.out.println("a может быть приведен к Object");
        if(b instanceof Object)
            System.out.println("b может быть приведен к Object");
        if(c instanceof Object)
            System.out.println("c может быть приведен к Object");
        if(d instanceof Object)
            System.out.println("d может быть приведен к Object");
    }
}
```

Результат работы этой программы таков.

```
a есть экземпляр A
b есть экземпляр B
c есть экземпляр C
c может быть приведен к A

ob теперь ссылается на d
ob есть экземпляр D

ob теперь ссылается на c
ob не может быть приведен к D
ob может быть приведено к A

a может быть приведен к Object
b может быть приведен к Object
c может быть приведен к Object
d может быть приведен к Object
```

Большинство программ не нуждается в операторе `instanceof`, поскольку обычно вам известны типы объектов, с которыми вы работаете. Однако он может оказаться очень полезным, когда вы разрабатываете обобщенные процедуры, имеющие дело с объектами из сложной иерархии классов.

Модификатор `strictfp`

Этот модификатор является относительно новым ключевым словом. Когда был выпущен язык Java 2, модель вычислений с плавающей точкой была слегка упро-

щена. В частности, новая модель не требовала округления некоторых промежуточных результатов вычислений. В ряде случаев это предотвращает переполнение. Модифицируя класс, метод или интерфейс ключевым словом `strictfp`, вы гарантируете, что вычисления с плавающей точкой будут выполняться точно так, как они выполнялись в ранних версиях языка Java. Когда класс модифицирован словом `strictfp`, все его методы автоматически модифицируются как `strictfp`.

Например, следующий фрагмент сообщает Java, что нужно использовать исходную модель вычислений с плавающей точкой при вычислении всех методов, определенных в классе `MyClass`.

```
strictfp class MyClass { //...
```

Откровенно говоря, большинству программистов никогда не понадобится модификатор `strictfp`, поскольку он касается лишь небольшого класса проблем.

Машинно-зависимые методы

Хотя это случается редко, но все же иногда может понадобиться вызвать подпрограмму, написанную на языке, отличном от языка Java. Обычно такая подпрограмма существует в виде исполняемого кода для центрального процессора и среды, в которой вы работаете, то есть в виде машинно-зависимого (native) кода. Например, вы можете решить вызвать такую подпрограмму для повышения скорости выполнения. Или же вам может понадобиться работать со специализированной библиотекой от независимых поставщиков, например с пакетом статистических расчетов. Однако поскольку программы Java компилируются в код виртуальной машины, который затем интерпретируется (или компилируется "на лету") исполняющей системой Java, вызов подпрограмм машинно-зависимого кода из программ на языке Java может показаться невозможным. К счастью, это заключение ложно. В языке Java предусмотрено ключевое слово `native`, которое используется для объявления машинно-зависимых методов. Однажды объявленные, эти методы могут быть вызваны из вашей программы Java точно так же, как вызывается любой другой метод Java.

Чтобы объявить машинно-зависимый метод, предварите его имя модификатором `native`, но не определяйте тело метода.

```
public native int meth() ;
```

После объявления метода нужно собственно написать его и выполнить серию относительно сложных шагов, чтобы соединить его с кодом Java.

Большинство машинно-зависимых методов пишется на языке C. Механизм интеграции кода C с программой Java называется интерфейсом JNI (Java Native Interface). Подробное описание JNI выходит за рамки настоящей книги, но предложенное ниже краткое описание дает достаточную информацию для большинства приложений.

На заметку! Конкретные действия, которые следует предпринять, зависят от используемой среды Java. Они также зависят от языка, который используется для реализации машинно-зависимых методов. Следующий пример ориентирован на среду Windows. Язык реализации метода — C.

Простейший способ понять процесс — исследовать его на примере. Для начала введите следующую короткую программу, которая использует машинно-зависимый метод по имени `test()`.

```
// Простой пример использования машинно-зависимого метода.
public class NativeDemo {
    int i;
    public static void main(String args[]) {
        NativeDemo ob = new NativeDemo();

        ob.i = 10;
        System.out.println("Это ob.i перед вызовом
                           машинно-зависимого метода:" + ob.i);
        ob.test(); // вызов native метода
        System.out.println("Это ob.i после вызова
                           машинно-зависимого метода:" + ob.i);
    }

    // Объявление машинно-зависимого метода
    public native void test() ;

    // загрузить библиотеку DLL, содержащую статический метод
    static {
        System.loadLibrary("NativeDemo");
    }
}
```

Обратите внимание на то, что метод `test()` объявлен как `native` и не имеет тела. Это метод, который будет вскоре реализован на языке C. Также посмотрите на статический блок. Как уже упоминалось ранее, блок, объявленный как `static`, выполняется только однажды — при запуске программы (или, точнее говоря, при первой загрузке ее класса). В этом случае он используется для загрузки динамической библиотеки, которая содержит реализацию метода `test()`. (Вскоре вы увидите, как создать такую библиотеку.)

Библиотека загружается методом `loadLibrary()`, который является частью класса `System`. Его общая форма такова.

```
static void loadLibrary(String имяФайла)
```

Здесь *имяФайла* — строка, которая задает имя файла, содержащего библиотеку. Для среды Windows предполагается, что файл имеет расширение `.DLL`.

После ввода текста программы скомпилируйте ее, чтобы получить файл `NativeDemo.class`. Далее следует использовать приложение `javah.exe`, чтобы создать один заголовочный файл — `NativeDemo.h` (приложение `javah.exe` включено в комплект JDK). Вы включите файл `NativeDemo.h` в свою реализацию метода `test()`. Чтобы получить файл `NativeDemo.h`, выполните следующую команду.

```
javah -jni NativeDemo
```

Эта команда создает файл по имени `NativeDemo.h`. Этот файл должен быть включен в файл C, реализующий метод `test()`. Вывод, созданный этой командой, показан ниже.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class NativeDemo */

#ifndef _Included_NativeDemo
#define _Included_NativeDemo
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class: NativeDemo
 * Method: test
```

```
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Обратите особое внимание на следующую строку, которая определяет прототип создаваемой вами функции `test()`.

```
JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *, jobject);
```

Отметим, что именем функции будет `Java_NativeDemo_test()`. Его и следует использовать в качестве имени машинно-зависимой функции, которую вы реализуете. То есть вместо написания на языке C функции `test()` вы создаете функцию `Java_NativeDemo_test()`. Часть `NativeDemo` в префиксе добавляется, поскольку она указывает, что метод `test()` является членом класса `NativeDemo`. Помните, что другой класс может объявить свой собственный метод `test()`, абсолютно отличный от того, что объявлен в классе `NativeDemo`. Включение имени класса в префикс позволяет различать версии. Основное правило: машинно-зависимым функциям присваивается имя, префикс которого включает имя класса, в котором он объявлен.

После создания необходимого заголовочного файла вы можете написать свою реализацию метода `test()` и сохранить ее в файле `NativeDemo.c`.

```
/* Этот файл содержит C-версию метода test().*/

#include <jni.h>
#include "NativeDemo.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *env, jobject obj)
{
    jclass cls;
    jfieldID fid;
    jint i;

    printf("Запуск машинно-зависимого метода.\n");
    cls = (*env)->GetObjectClass(env, obj);
    fid = (*env)->GetFieldID(env, cls, "i", "I");

    if(fid == 0) {
        printf("Невозможно получить поле id.\n");
        return;
    }
    i = (*env)->GetIntField(env, obj, fid);
    printf("i = %d\n", i);
    (*env)->SetIntField(env, obj, fid, 2*i);
    printf("Завершение машинно-зависимого метода.\n");
}
```

Отметим, что этот файл включает заголовок `jni.h`, содержащий интерфейсную информацию. Этот файл поставляется вместе с компилятором Java. Напомним, что заголовок `NativeDemo.h` ранее создан командой `javah`.

В этой функции метод `GetObjectClass()` используется для получения структуры C, имеющей информацию о классе `NativeDemo`. Метод `GetFieldID()` возвращает структуру C с информацией о поле класса по имени `i`. Метод `GetIntField()`

извлекает исходное значение этого поля и сохраняет обновленное значение этого поля (см. в файле `jni.h` дополнительные методы, которые управляют другими типами данных).

После создания файла `NativeDemo.c` его следует скомпилировать и создать библиотеку DLL. Чтобы сделать это с помощью компилятора Microsoft C/C++, используйте следующую командную строку (возможно, понадобится указать путь к файлу `jni.h` и его подчиненному файлу `jni_md.h`).

```
Cl /LD NativeDemo.c
```

Эта команда создаст файл `NativeDemo.dll`. Только после того как все будет сделано, вы сможете запустить программу Java, которая выдаст такой результат.

```
Это ob.i перед вызовом машинно-зависимого метода: 10
Запуск машинно-зависимого метода.
i = 10
Завершение машинно-зависимого метода.
Это ob.i после вызова машинно-зависимого метода: 20
```

Проблемы, связанные с машинно-зависимыми методами

Машинно-зависимые методы выглядят многообещающе, поскольку позволяют получить доступ к существующей базе библиотечных подпрограмм, а также надеяться на высокую скорость работы программ. Однако с этими методами связаны две существенные проблемы.

- **Потенциальный риск нарушения безопасности**. Поскольку машинно-зависимый метод выполняет реальный машинный код, он может получить доступ к любой части системы. То есть машинно-зависимый код не относится к исполняющей среде Java. Это, например, угрожает вирусной инфекцией. По этой причине аплеты не могут использовать машинно-зависимые методы. Кроме того, загрузка библиотеки DLL может быть ограничена, и она может быть субъектом утверждения для менеджера по безопасности.
- **Потеря переносимости**. Поскольку машинно-зависимый код содержится в библиотеке DLL, он должен быть представлен на машине, которая выполняет программу Java. Более того, поскольку каждый машинно-зависимый метод зависит от процессора и операционной системы, каждая библиотека DLL, как следствие, не является переносимой. То есть приложение Java, которое использует машинно-зависимые методы, сможет выполняться только на машине, на которой установлена совместимая библиотека DLL.

Применение машинно-зависимых методов должно быть ограничено, поскольку они делают вашу программу Java непереносимой и представляют существенный риск нарушения безопасности.

Использование ключевого слова `assert`

Еще одним относительно новым дополнением к языку Java является ключевое слово `assert`. Оно используется во время разработки программ для создания так называемых *утверждений* (assertion), представляющих собой условия, которые должны быть истинными во время выполнения программы. Например, у вас мо-

жет быть метод, который всегда возвращает положительное целое значение. Вы можете проверить его утверждением, что возвращаемое значение больше нуля, используя оператор `assert`. Если во время выполнения условие истинно, то никаких других действий не выполняется. Однако если условие окажется ложным, будет передано исключение `AssertionError`. Утверждения часто применяются при верификации того, что некоторое ожидаемое условие действительно выполняется. В коде окончательной версии они, как правило, отсутствуют.

Ключевое слово `assert` имеет две формы. Первая выглядит так.

```
assert условие;
```

Здесь *условие* — выражение, которое должно при вычислении дать булев результат. Если результат равен `true`, то утверждение истинно и никаких действий не выполняется. Если же условие дает `false`, значит, произошел сбой и передается объект исключения по умолчанию `AssertionError`.

Вторая форма оператора `assert` выглядит следующим образом.

```
assert условие: выражение;
```

В этой версии *выражение* — значение, которое передается конструктору исключения `AssertionError`. Это значение преобразуется в строковую форму и отображается, если утверждение ложно. Обычно вы задаете строку для *выражение*, но разрешено любое выражение, отличное от `void`, до тех пор, пока оно допускает осмысленное строковое преобразование.

Ниже показан пример использования оператора `assert`. В нем осуществляется проверка того, что возвращаемое значение метода `getnum()` положительно.

```
// Демонстрация assert.
class AssertDemo {
    static int val = 3;

    // Возвращает целочисленное значение.
    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; // произойдет сбой, если n == 0

            System.out.println("n равно " + n);
        }
    }
}
```

Чтобы включить проверку утверждений во время выполнения, следует указать параметр `-ea`. Например, чтобы сделать это для класса `AssertDemo`, выполните следующую команду.

```
java -ea AssertDemo
```

После компиляции и запуска, как показано выше, программа выдает следующий результат.

```
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" java.lang.AssertionError
в AssertDemo.main(AssertDemo.java:17)
```

В методе `main()` выполняются повторяющиеся вызовы метода `getnum()`, который возвращает целочисленное значение. Возвращаемое значение метода `getnum()` присваивается переменной `n`, а затем проверяется оператором `assert`.

```
assert n > 0; // произойдет сбой, если n == 0
```

Этот оператор завершится сбоем, когда значение переменной `n` будет равно нулю, что произойдет после четвертого вызова. Когда подобное случится, будет передано исключение.

Как объяснялось, вы можете задать сообщение, отображаемое при сбое утверждения. Например, если вы подставите

```
assert n > 0 : "n отрицательное!";
```

в утверждение из предыдущей программы, то будет выдан такой результат.

```
n равно 3
n равно 2
n равно 1
Exception in thread "main" java.lang.AssertionError : n отрицательное!
at AssertDemo.main(AssertDemo.java:17)
```

Один момент, важный для понимания утверждений, – это то, что вы не должны полагаться на них для выполнения каких-либо действий программы. Причина в том, что нормальный код окончательной версии будет выполняться с отключенным механизмом проверки утверждений. Например, рассмотрим следующий вариант предыдущей программы.

```
// Плохой способ применения assert!!!
class AssertDemo {
    // получить генератор случайных чисел
    static int val = 3;

    // Возвращает целое.
    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; // Плохая идея!

            System.out.println("n is " + n);
        }
    }
}
```

В этой версии программы вызов метода `getnum()` перемещен в оператор `assert`. Хотя это хорошо работает, когда механизм проверки утверждений включен, его отключение приведет к неправильной работе программы, потому что вызов метода `getnum()` никогда не произойдет! Фактически значение переменной `n` теперь должно быть инициализировано, поскольку компилятор распознает ситуацию, что значение может не быть присвоено в операторе `assert`.

Утверждения — хорошее нововведение в язык Java, потому что оно упрощает тип проверки ошибок, который часто используется во время разработки. Так, например, если до появления утверждений вы хотели проверить, что переменная n имеет положительное значение в приведенной выше программе, то должны были написать примерно следующую последовательность кода.

```
if(n < 0) {
    System.out.println("n отрицательное!");
    return; // или передать исключение
}
```

Для применения утверждения нужна только одна строка кода. Более того, вам не придется удалять строки утверждений из окончательного варианта кода.

Параметры включения и отключения утверждений

При выполнении кода вы можете отключить все утверждения параметром -da. Вы можете включить или отключить его для специфического пакета (и всех его внутренних пакетов), указав его имя, три точки и параметр -ea или -da. Например, чтобы включить механизм проверки утверждений для пакета MyPack, используйте следующее.

```
-ea:MyPack...
```

Для того чтобы отключить такой механизм проверки утверждений, примените следующее.

```
-da:MyPack...
```

Вы можете также задать класс с параметром -ea или -da. Например, это включает индивидуально класс AssertDemo.

```
-ea:AssertDemo
```

Статический импорт

Язык Java имеет такое средство, как *статический импорт* (static import), которое расширяет возможности ключевого слова import. Ключевое слово import с предшествующим ключевым словом static может применяться для импорта статических членов класса или интерфейса. При использовании статического импорта появляется возможность ссылаться на статические члены непосредственно по именам, без необходимости квалифицировать их именем класса. Это упрощает и сокращает синтаксис, необходимый для работы со статическими членами.

Чтобы понять удобство статического импорта, давайте начнем с примера, который *не* использует его. Следующая программа вычисляет гипотенузу прямоугольного треугольника. Она использует два статических метода из встроенного класса Java Math, входящего в пакет java.lang. Первый из них — Math.pow() — возвращает значение, возведенное в указанную степень. Второй — Math.sqrt() — возвращает квадратный корень аргумента.

```
// Вычисляет длину 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
        // квалифицированы именем их класса - Math.
        hypot = Math.sqrt(Math.pow(side1, 2) +
                          Math.pow(side2, 2));

        System.out.println("Даны длины сторон " +
                           side1 + " и " + side2 +
                           " гипотенуза равна " + hypot);
    }
}
```

Поскольку методы `pow()` и `sqrt()` — статические, они должны быть вызваны с указанием имени их класса — `Math`. Это приводит к следующему громоздкому вычислению гипотенузы.

```
hypot = Math.sqrt(Math.pow(side1, 2) +
                  Math.pow(side2, 2));
```

Как иллюстрирует этот простой пример, довольно утомительно каждый раз указывать имя класса при вызовах методов `pow()` и `sqrt()` (или любых других математических методов Java вроде `sin()`, `cos()` и `tan()`).

Вы можете избежать утомительного повторения имени пакета благодаря применению статического импорта, как показано в следующей версии предыдущей программы.

```
// Применение статического импорта, делающего sqrt() и pow() видимыми.
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

// Вычисление гипотенузы прямоугольного треугольника.
class Hypot {
    public static void main(String args[]) {
        double side1, side2;
        double hypot;

        side1 = 3.0;
        side2 = 4.0;

        // Здесь sqrt() ирow() можно вызывать
        // непосредственно, без их имени класса.
        hypot = sqrt(pow(side1, 2) + pow(side2, 2));

        System.out.println("Даны длины сторон " +
                           side1 + " и " + side2 +
                          " гипотенуза равна " + hypot);
    }
}
```

В этой версии имена `sqrt` и `pow` становятся видимыми благодаря оператору статического импорта.

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

После этих операторов больше нет необходимости квалифицировать имена методов `pow()` и `sqrt()` именем их класса. Таким образом, вычисление гипотенузы может быть выражено более удобно.

```
hypot = sqrt(pow(side1, 2) + pow(side2, 2));
```

Как видите, эта форма и более читабельна.

Существует две основные формы оператора `import static`. Первая, которая использовалась в предыдущем примере, делает видимым единственное имя. Его общая форма такова.

```
import static пакет.имя_типа.имя_статического_члена;
```

Здесь *имя_типа* — имя класса или интерфейса, который содержит требуемый статический член. Полное имя его пакета указано в части *пакет*, а имя члена — в *имя_статического_члена*.

Вторая форма статического импорта позволяет импортировать все статические члены данного класса или интерфейса. Его общая форма показана ниже.

```
import static пакет.имя_типа.*;
```

Если вы будете использовать много статических методов или полей, определенных в классе, то эта форма позволит вам сделать их видимыми без необходимости указывать каждый отдельно. Таким образом, в предыдущей программе с помощью единственного оператора `import` можно ввести в область видимости методы `pow()` и `sqrt()` (а также *все другие* статические члены класса `Math`).

```
import static java.lang.Math.*;
```

Конечно же, статический импорт не ограничивается только классом `Math` или его методами. Например, следующая строка вводит в область видимости статическое поле `System.out`.

```
import static java.lang.System.out;
```

После этого оператора вы можете выводить информацию на консоль потоком `out`, не указывая его класс `System`, как показано здесь.

```
out.println("Импортировав System.out, вы можете использовать его
непосредственно.");
```

Однако импортировать переменную-член `System.out`, как показано выше, — это не только хорошая идея, но и предмет обсуждения. Несмотря на то что это сокращает текст программы, все же теперь не будет очевидно тем, кто читает программу, что `out` относится к переменной-члену `System.out`.

Еще один момент: в дополнение к импорту статических членов классов и интерфейсов, определенных в Java API, вы можете также использовать статический импорт для импортирования статических членов ваших собственных классов и интерфейсов.

Каким бы удобным ни казался статический импорт, важно не злоупотреблять им. Помните, что причина объединения библиотечных классов Java в пакеты позволяет избежать конфликтов пространств имен и непреднамеренного сокрытия прочих имен. Если вы используете в своей программе статический член однажды или дважды, лучше его не импортировать. К тому же некоторые статические имена, как, например `System.out`, настолько привычны и узнаваемы, что, вероятно, вы вообще не захотите импортировать их. Статический импорт предназначен для тех ситуаций, когда вы применяете статические члены многократно, как, например, при выполнении серии математических вычислений. То есть, в сущности, вам стоит использовать это средство, но не злоупотреблять им.

Вызов перегруженных конструкторов через `this()`

Имея дело с перегруженными конструкторами, иногда удобно один конструктор вызывать из другого. В языке Java это обеспечивается использованием другой формы ключевого слова `this`. Вот его общая форма.

```
this(список_аргументов)
```

При выполнении конструктора `this()` сначала выполняется перегруженный конструктор, который соответствует списку параметров *список_аргументов*. Затем выполняются операторы, находящиеся внутри исходного конструктора, если таковые присутствуют. Вызов конструктора `this()` должен быть первым оператором в конструкторе.

Чтобы понять, как следует использовать конструктор `this()`, рассмотрим короткий пример. Для начала приведем класс, который *не* использует конструктор `this()`.

```
class MyClass {
    int a;
    int b;

    // Инициализировать a и b индивидуально
    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;
    }

    // Присвоить a и b значение по умолчанию 0
    MyClass( ) {
        a = 0;
        b = 0;
    }
}
```

Этот класс включает в себя три конструктора, каждый из которых инициализирует значения переменных `a` и `b`. Первому передаются индивидуальные значения для переменных `a` и `b`. Второй принимает только одно значение и присваивает его переменным `a` и `b`. Третий присваивает переменным `a` и `b` значение по умолчанию — 0.

Используя конструктор `this()`, можно переписать приведенный класс `MyClass` следующим образом.

```
class MyClass {
    int a;
    int b;

    // Инициализировать a и b индивидуально
    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);
    }

    // Присвоить a и b значение по умолчанию 0
    MyClass( ) {
        this(0); // вызывается MyClass(0)
    }
}
```

В этой версии класса `MyClass` единственным конструктором, который в действительности присваивает значения полям `a` и `b`, является `MyClass(int, int)`. Например, посмотрим, что случится при выполнении следующего оператора.

```
MyClass mc = new MyClass(8);
```

Вызов конструктора `MyClass(8)` приводит к выполнению конструктора `this(8,8)`, что транслируется в вызов конструктора `MyClass(8,8)`, поскольку именно эта версия конструктора класса `MyClass` соответствует данному вызову конструктора `this()` по списку параметров. Теперь рассмотрим следующий оператор, использующий конструктор по умолчанию.

```
MyClass mc2 = new MyClass();
```

В этом случае вызывается конструктор `this(0)`, что приводит к выполнению конструктора `MyClass(0)`, поскольку именно эта версия конструктора подходит по списку параметров. Конечно же, конструктор `MyClass(0)` затем обращается к конструктору `MyClass(0,0)`, как только что было описано.

Одна из причин, по которой стоит вызывать перегруженные конструкторы через конструктор `this()`, — исключение дублирования кода. Во многих случаях сокращение дублированного кода уменьшает время загрузки классов, поскольку уменьшается объем кода объекта. Это особенно важно для программ, доставляемых по Интернету, когда время загрузки критично. Применение конструктора `this()` может также помочь структурировать ваш код, когда конструкторы содержат большой объем дублированного кода.

Однако необходимо соблюдать осторожность. Конструкторы, которые вызывают конструктор `this()`, выполнятся немного медленнее, чем те, весь свой код инициализации которых содержится встроенным. Дело в том, что механизм вызова и возвращения, используемый при вызове второго конструктора, является дополнительной затратой. Если ваш класс будет использоваться для создания небольшого количества объектов или если конструктор, вызывающий конструктор `this()` будет использоваться редко, то снижение производительности во время выполнения, вероятно, будет незначительным. Но если во время выполнения программы предполагается создание большого количества объектов вашего класса (порядка тысяч), то негативное воздействие увеличения дополнительных затрат может оказаться значительным. Поскольку создание объектов затрагивает всех пользователей вашего класса, вам придется тщательно взвесить преимущества более быстрой загрузки по сравнению с увеличением времени на создание объекта.

Еще одно замечание: для очень коротких конструкторов, таких как в классе `MyClass`, зачастую различие в размере объектного кода с использованием конструктора `this()` или без него небольшое. (Фактически в некоторых случаях никакого уменьшения размера объектного кода нет.) Дело в том, что битовый код, который устанавливается и возвращается из вызова конструктора `this()`, добавляет инструкции к объектному файлу. Поэтому в таких ситуациях, несмотря на устранение дублирования кода, использование конструктора `this()` не даст существенной экономии времени загрузки. Однако дополнительные затраты на создание каждого объекта все еще возможны. Поэтому применение конструктора `this()` наиболее подходит к конструкторам, которые содержат большие объемы кода инициализации, а не те, которые просто устанавливают значения нескольких полей.

При использовании конструктора `this()` следует учитывать следующее. Во-первых, в вызове конструктора `this()` вы не можете использовать переменные экземпляра класса конструктора. Во-вторых, вы не можете использовать конструкторы `super()` и `this()` в том же конструкторе, поскольку вызов каждого из них должен быть первым оператором в конструкторе.

ГЛАВА 14 Обобщения

После выхода в 1995 году первоначальной версии 1.0 в язык Java было добавлено множество новых средств. Одним из наиболее значительных и влиятельных новшеств стали *обобщения* (generic). Во-первых, их появление означало добавление новых синтаксических элементов в язык. Во-вторых, они повлекли за собой изменения во многих классах и методах самого ядра API. Сегодня обобщения — это неотъемлемая часть программирования на языке Java, и твердое понимание этого важного средства обязательно. Здесь оно исследуется подробно.

Применение обобщений позволило создавать классы, интерфейсы и методы, работающие безопасным к типам способом с разнообразными видами данных. Многие алгоритмы логически идентичны, независимо от того, к данным каких типов они применяются. Например, механизм, поддерживающий стеки, является одним и тем же в стеках, хранящих элементы классов `Integer`, `String`, `Object` или `Thread`. Благодаря обобщениям, вы можете определить алгоритм однажды, независимо от конкретного типа данных, а затем применять его к широкому разнообразию типов данных без каких-либо дополнительных усилий. Впечатляющая мощь добавленных к языку обобщений фундаментально изменила способы написания кода Java.

Вероятно, одно из средств Java, которое в наибольшей степени испытало влияние обобщений, — это инфраструктура *коллекций* Collections Framework. Упомянутая инфраструктура является частью API Java и подробно описана в главе 17, но стоит вкратце пояснить ее здесь. *Коллекция* — это группа объектов. Инфраструктура коллекций определяет несколько классов, таких как списки и карты, которые управляют коллекциями. Классы коллекций всегда готовы работать с объектами любых типов. Выгода от добавления в язык обобщений состоит в том, что классы коллекций теперь могут использоваться с полным обеспечением безопасности типов. То есть, помимо предоставления мощного элемента языка, обобщения также значительно усовершенствовали существующие средства. Вот почему обобщения представляют собой столь значимое дополнение к языку Java.

В настоящей главе описаны синтаксис, теория и применение обобщений. Мы покажем, как обобщения обеспечивают безопасность типов в некоторых ранее трудных случаях. Когда вы изучите эту главу, то захотите ознакомиться с главой 17, описывающей систему коллекций. Там вы найдете множество примеров работы обобщений.

Помните! Обобщения были добавлены в J2SE 5.0. Использующий их код не может быть скомпилирован старыми версиями `javac`.

Что такое обобщения

По сути дела, *обобщения* — это *параметризованные типы*. Эти типы важны, поскольку позволяют объявлять классы, интерфейсы и методы, где тип данных, ко-

торыми они оперируют, указан в виде параметра. Используя обобщения, можно создать единственный класс, который, например, будет автоматически работать с разными типами данных. Классы, интерфейсы или методы, имеющие дело с параметризованными типами, называются *обобщениями*, *обобщенными классами* или *обобщенными методами*.

Важно понимать, что язык Java всегда предлагал возможность создавать в определенной мере обобщенные классы, интерфейсы и методы, оперирующие ссылками на тип `Object`. Поскольку тип `Object` — это суперкласс для всех остальных классов, ссылка на тип `Object` может обращаться к объекту любого типа. То есть в старом коде обобщенные классы, интерфейсы и методы использовали ссылки на тип `Object` для того, чтобы оперировать объектами различного типа. Проблема была в том, что они не могли обеспечить безопасность типов.

Обобщения добавили в язык безопасность типов, которой так не хватало. Они также упростили процесс выполнения, поскольку теперь нет необходимости применять явные приведения для транслирования объектов класса `Object` в реальные типы данных, с которыми выполняются действия. Благодаря обобщениям, все приведения выполняются автоматически и неявно. То есть обобщения расширили ваши возможности повторного использования кода и позволили вам делать это легко и безопасно.

На заметку! Предупреждение для программистов C++: хотя обобщения похожи на шаблоны в C++, это не одно и то же. Существует ряд фундаментальных отличий между двумя подходами к обобщенным типам. Если у вас имеется опыт применения языка C++, важно не делать поспешных выводов о том, как обобщения работают в языке Java.

Простой пример обобщения

Давайте начнем с простого примера обобщенного класса. В следующей программе определены два класса. Первый — это обобщенный класс `Gen`, а второй — `GenDemo`, класс использующий класс `Gen`.

```
// Простой обобщенный класс.
// Здесь T - это параметр типа,
// который будет заменен реальным типом
// при создании объекта класса Gen.
class Gen<T> {
    T ob; // объявление объекта типа T

    // Передать конструктору ссылку
    // на объект типа T.
    Gen(T o) {
        ob = o;
    }

    // Вернуть ob.
    T getob() {
        return ob;
    }

    // Показать тип T.
    void showType() {
        System.out.println("Типом T является " + ob.getClass().getName());
    }
}
```

```
// Демонстрация обобщенного класса.
class GenDemo {
    public static void main(String args[]) {
        // Создать Gen-ссылку для Integers.
        Gen<Integer> iOb;

        // Создать объект Gen<Integer> и присвоить
        // ссылку на iOb. Отметьте применение автоупаковки
        // для инкапсуляции значения 88 в объект Integer.
        iOb = new Gen<Integer>(88);

        // Показать тип данных, используемый iOb.
        iOb.showType();

        // Получить значение iOb. Обратите внимание,
        // что никакого приведения не нужно.
        int v = iOb.getob();
        System.out.println("значение: " + v);
        System.out.println();

        // Создать объект Gen для String.
        Gen<String> strOb = new Gen<String> ("Обобщенный тест");

        // Показать тип данных, используемый strOb.
        strOb.showType();

        // Получить значение strOb. Опять же
        // приведение не требуется.
        String str = strOb.getob();
        System.out.println("Значение: " + str);
    }
}
```

Результат работы этой программы.

```
Типом T является java.lang.Integer
Значение: 88

Типом T является java.lang.String
Значение: Обобщенный тест
```

Давайте внимательно исследуем эту программу. Обратите внимание на объявление класса `Gen` в следующей строке.

```
class Gen<T> {
```

Здесь `T` — имя *параметра типа*. Это имя используется в качестве заполнителя, куда будет подставлено имя реального типа, переданного классу `Gen` при создании реальных типов. То есть параметр типа `T` применяется в классе `Gen` всякий раз, когда требуется параметр типа. Обратите внимание на то, что тип `T` заключен в угловые скобки <>. Этот синтаксис может быть обобщен. Всякий раз, когда объявляется параметр типа, он указывается в угловых скобках. Поскольку класс `Gen` применяет параметр типа, класс `Gen` является обобщенным классом, который называется также *параметризованным типом*.

Далее тип `T` используется для объявления объекта по имени `ob`, как показано ниже.

```
T ob; // объявляет объект типа T
```

Как упоминалось, `T` — это место для подстановки реального типа, который будет указан при создании объекта класса `Gen`. То есть объект `ob` будет объектом

типа, переданного в параметре типа T. Например, если в параметре T передан тип String, то экземпляр ob будет иметь тип String.

Теперь рассмотрим конструктор Gen().

```
Gen(T o) {
    ob = o;
}
```

Как видите, параметр o имеет тип T. Это значит, что реальный тип параметра o определяется типом, переданным параметром типа T при создании объекта класса Gen. К тому же, поскольку и параметр o, и переменная-член ob имеют тип T, они оба получают одинаковый реальный тип при создании объекта класса Gen.

Параметр типа T также может быть использован для указания типа возвращаемого значения метода, как в случае метода getob(), показанного здесь.

```
T getob() {
    return ob;
}
```

Так как объект ob тоже имеет тип T, его тип совместим с типом, возвращаемым методом getob().

Метод showType() отображает тип T вызовом метода getName() объекта класса Class, возвращенным вызовом метода getClass() объекта ob. Метод getClass() определен в классе Object и потому является членом всех классов. Он возвращает объект класса Class, соответствующий типу класса объекта, для которого он вызван. Класс Class определяет метод getName(), который возвращает строковое представление имени класса.

Класс GenDemo демонстрирует обобщенный класс Gen. Сначала он создает версию класса Gen для целых чисел, как показано ниже.

```
Gen<Integer> iOb;
```

Посмотрим на это объявление внимательней. Отметим, что тип Integer указан в угловых скобках после слова Gen. В этом случае Integer — это аргумент типа, который передается в параметре типа T класса Gen. Это фактически создает версию класса Gen, в которой все ссылки на тип T транслируются в ссылки на тип Integer. То есть в данном объявлении объект ob имеет тип Integer, и тип возвращаемого значения метода getob() также имеет тип Integer.

Прежде чем двигаться дальше, необходимо заявить, что компилятор Java на самом деле не создает различные версии класса Gen или любого другого обобщенного класса. Хотя было бы удобно считать так, на самом деле подобное не происходит. Вместо этого компилятор удаляет всю обобщенную информацию о типах, выполняя необходимые приведения, чтобы сделать поведение вашего кода таким, будто создана специфическая версия класса Gen. То есть имеется только одна версия класса Gen, которая существует в вашей программе. Процесс удаления обобщенной информации о типе называется *очисткой* (erasure), и мы вернемся к этой теме чуть позднее в настоящей главе.

Следующая строка присваивает объекту iOb ссылку на экземпляр целочисленной версии класса Gen.

```
iOb = new Gen<Integer>(88);
```

Отметим, что когда вызывается конструктор Gen(), аргумент типа Integer также указывается. Это необходимо, потому что типом объекта (в данном случае объекта iOb), которому присваивается ссылка, является тип Gen<Integer>. То есть ссылка, возвращаемая оператором new, также должна иметь тип Gen<Integer>.

Если это не так, получается ошибка времени компиляции. Например, следующее присваивание вызовет ошибку компиляции.

```
iOb = new Gen<Double>(88.0); // Ошибка!
```

Поскольку объект iOb имеет тип Gen<Integer>, он не может быть использован для присваивания ссылки типа Gen<Double>. Эта проверка типа является одним из основных преимуществ обобщений, потому что обеспечивает безопасность типов.

Как указано в комментарии к программе, присваивание

```
iOb = new Gen<Integer>(88);
```

использует автоупаковку для инкапсуляции значения 88, имеющего тип int, в объекте класса Integer. Это работает, потому что тип Gen<Integer> создает конструктор, принимающий аргумент класса Integer. Поскольку ожидается объект класса Integer, Java автоматически упаковывает 88 внутрь него. Конечно, присваивание также может быть написано явно, как здесь.

```
iOb = new Gen<Integer>(new Integer(88));
```

Однако с этой версией не связано никаких преимуществ.

Программа затем отображает тип объекта ob внутри объекта iOb, которым является Integer. Далее программа получает значение объекта ob в следующей строке.

```
int v = iOb.getob();
```

Поскольку возвращаемым типом метода getob() будет T, который заменяется на Integer при объявлении объекта iOb, то возвращаемым типом метода getob() также будет класс Integer, который автоматически распаковывается в тип int и присваивается переменной v, имеющей тип int. То есть нет никакой необходимости приводить тип возвращаемого значения метода getob() к классу Integer. Конечно, использовать автоупаковку не обязательно. Предыдущая строка может быть написана так.

```
int v = iOb.getob().intValue();
```

Однако автоупаковка позволяет сделать код более компактным.

Далее в классе GenDemo объявляется объект типа Gen<String>.

```
Gen<String> strOb = new Gen<String>(" Обобщенный тест");
```

Поскольку аргументом типа является String, класс String подставляется вместо параметра T внутри класса Gen. Это создает (концептуально) строковую версию класса Gen, что и демонстрируют остальные строки программы.

Обобщения работают только с объектами

Когда объявляется экземпляр обобщенного типа, аргумент, переданный в качестве параметра типа, должен быть типом класса. Вы не можете использовать элементарный тип вроде int или char. Например, классу Gen можно передать в параметре T любой тип класса, но нельзя передать элементарный тип. Таким образом, следующее объявление недопустимо.

```
Gen<int> intOb = new Gen<int>(53); // Ошибка, нельзя использовать
                                   // элементарные типы
```

Конечно, невозможность использовать элементарный тип не является серьезным ограничением, так как вы можете применять оболочки типов (как это и делается в предыдущем примере) для инкапсуляции элементарных типов. Более того, механизм автоупаковки и автораспаковки Java делает использование оболочек типов прозрачным.

Отличие обобщенных типов в зависимости от аргументов типа

Ключевой момент в понимании обобщенных типов в том, что ссылка на одну специфическую версию обобщенного типа не совместима с другой версией того же обобщенного типа. Например, следующая строка, если ее добавить к предыдущей программе, вызовет ошибку и программа не будет откомпилирована.

```
iOb = strOb; // Не верно!
```

Даже несмотря на то, что объекты `iOb` и `strOb` имеют тип `Gen<T>`, они являются ссылками на разные типы, потому что типы их параметров отличаются. Это часть того способа, благодаря которому обобщения обеспечивают безопасность типов и предотвращают ошибки.

Обобщения повышают безопасность типов

Теперь вы можете задать себе следующий вопрос: если те же функциональные возможности, которые мы обнаружили в обобщенном классе `Gen`, могут быть получены без обобщений, т.е. простым указанием класса `Object` в качестве типа данных и применением правильных приведений, то в чем же выгода от того, что класс `Gen` параметризован? Ответ: в том, что обобщения автоматически гарантируют безопасность типов во всех операциях, где задействован класс `Gen`. В процессе работы с ним исключается необходимость явного приведения и ручной проверки типов в коде.

Чтобы понять выгоды от обобщений, для начала рассмотрим следующую программу, которая создает необобщенный эквивалент класса `Gen`.

```
// NonGen — функциональный эквивалент Gen,
// не использующий обобщений.
class NonGen {
    Object ob; // ob теперь имеет тип Object

    // Передать конструктору ссылку на объект типа Object
    NonGen(Object o) {
        ob = o;
    }

    // Вернуть тип Object.
    Object getob() {
        return ob;
    }

    // Показать тип ob.
    void showType() {
        System.out.println("Типом ob является " +
                           ob.getClass().getName());
    }
}
```

```
// Демонстрация необобщенного класса.
class NonGenDemo {
    public static void main(String args[]) {
        NonGen iOb;

        // Создать объект NonGen и сохранить
        // Integer в нем. Автоупаковка используется.
        iOb = new NonGen(88);

        // Показать тип данных, используемый iOb.
        iOb.showType();

        // Получить значение iOb.
        // На этот раз приведение необходимо.
        int v = (Integer) iOb.getob();
        System.out.println("значение: " + v);
        System.out.println();

        // Создать другой объект NonGen и
        // сохранить в нем String.
        NonGen strOb = new NonGen("Тест без обобщений");

        // Показать тип данных, используемый strOb.
        strOb.showType();

        // Получить значение strOb.
        // Опять же - приведение необходимо.
        String str = (String) strOb.getob();
        System.out.println("Значение: " + str);

        // Это компилируется, но концептуально неверно!
        iOb = strOb;
        v = (Integer) iOb.getob(); // ошибка времени выполнения!
    }
}
```

В этой версии программы присутствует несколько интересных моментов. Для начала класс `NonGen` заменяет все обращения к типу `T` на обращения к типу `Object`. Это позволяет классу `NonGen` хранить объекты любого типа, как это делает и обобщенная версия. Однако это не позволяет компилятору Java иметь какую-то реальную информацию о типе данных, в действительности сохраняемых в объекте класса `NonGen`, что плохо по двум причинам. Во-первых, для извлечения сохраненных данных требуется явное приведение. Во-вторых, многие ошибки несоответствия типов не могут быть обнаружены до времени выполнения. Рассмотрим каждую из этих проблем поближе.

Обратите внимание на эту строку.

```
int v = (Integer) iOb.getob();
```

Поскольку возвращаемым типом метода `getob()` является тип `Object`, необходимо привести его к типу `Integer`, чтобы позволить выполнить автораспаковку и сохранить значение в переменной `v`. Если убрать приведение, программа не компилируется. В версии с обобщением приведение происходит неявно. В версии без обобщения приведение должно быть явным. Это не только приносит неудобство, но и является потенциальным источником ошибок.

Теперь рассмотрим следующую кодовую последовательность в конце программы.

```
// Это компилируется, но концептуально неверно!
iOb = strOb;
v = (Integer) iOb.getob(); // ошибка времени выполнения!
```

Здесь объект `strOb` присваивается объекту `iOb`. Однако объект `strOb` ссылается на объект, содержащий строку, а не целое число. Это присваивание синтаксически корректно, потому что все ссылки класса `NonGen` одинаковы и любая ссылка класса `NonGen` может указывать на любой другой объект типа `NonGen`. Однако этот оператор семантически неверен, что и отражено в следующей строке. Здесь тип возвращаемого значения метода `getob()` приводится к классу `Integer`, а затем делается попытка присвоить это значение переменной `v`. Проблема в том, что объект `iOb` теперь ссылается на объект, который хранит тип `String`, а не `Integer`. К несчастью, без использования обобщений компилятор Java не имеет возможности обнаружить это. Вместо этого передается исключение времени выполнения. Возможность создавать безопасный в отношении типов код, в котором ошибки несоответствия типов перехватываются компилятором, — это главное преимущество обобщений. Хотя использование ссылок на тип `Object` для создания "псевдообобщенного" кода всегда возможно, нужно помнить, что такой код не является безопасным в отношении типов, и злоупотребление им приводит к исключениям времени выполнения. Обобщения предотвращают подобные вещи. По сути, благодаря обобщениям, ошибки времени выполнения преобразуются в ошибки времени компиляции. Это и есть главное преимущество.

Обобщенный класс с двумя параметрами типа

Для обобщенного типа можно объявлять более одного параметра типа. Чтобы указать два или более параметров типа, просто используйте разделенный запятыми список. Например, следующий класс `TwoGen` — это вариант класса `Gen`, который принимает два параметра.

```
// Простой обобщенный класс с двумя
// параметрами типа: T и V.
class TwoGen<T, V> {
    T ob1;
    V ob2;

    // Передать конструктору ссылки на объект типа T и объект типа V.
    TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }

    // Показать типы T и V.
    void showTypes() {
        System.out.println("Тип T: " + ob1.getClass().getName());

        System.out.println("Тип V: " + ob2.getClass().getName());
    }

    T getob1() {
        return ob1;
    }

    V getob2() {
        return ob2;
    }
```

```
}

// Демонстрация TwoGen.
class SimpGen {
    public static void main(String args[]) {

        TwoGen<Integer, String> tgObj =
            new TwoGen<Integer, String>(88, "Обобщения");

        // Показать типы.
        tgObj.showTypes();

        // Получить и показать значения.
        int v = tgObj.getob1();
        System.out.println("Значение: " + v);

        String str = tgObj.getob2();
        System.out.println("Значение: " + str);
    }
}
```

Результат работы этой программы.

```
Тип T: java.lang.Integer
Тип V: java.lang.String
Значение: 88
Значение: Обобщения
```

Обратите внимание на объявление класса `TwoGen`.

```
class TwoGen<T, V> {
```

Оно задает два параметра типа — `T` и `V`, — разделенные запятой. Поскольку оно имеет два параметра типа, при создании объекта класса `TwoGen` должны быть переданы два аргумента типа, как показано ниже.

```
TwoGen<Integer, String> tgObj =
    new TwoGen<Integer, String> (88, "Обобщения");
```

В этом случае класс `Integer` подставляется вместо параметра `T`, а класс `String` — вместо параметра `V`.

Хотя два аргумента в примере отличаются, допустимо передать в параметрах два одинаковых типа. Например, следующая строка кода вполне корректна.

```
TwoGen<String, String> x = new TwoGen<String, String> ("A", "B");
```

В этом случае оба аргумента параметров типа `V` и `T` будут иметь тип `String`. Конечно, если оба аргумента всегда будут одинаковы, то два параметра не обязательны.

Общая форма обобщенного класса

Синтаксис, показанный в предыдущих примерах, может быть обобщен. Так выглядит синтаксис объявления обобщенного класса.

```
class имя_класса<список_параметров_типа> { // ...
```

Ниже показан синтаксис объявления ссылки на обобщенный класс.

```
имя_класса<список_аргументов_типа> имя_переменной =
    new имя_класса<список_аргументов_типа> (список_аргументов_констант);
```

Ограниченные типы

В предыдущих примерах параметры типов могли быть заменены любыми типами классов. Это подходит ко многим случаям, но иногда удобно ограничить перечень типов, передаваемых в параметрах. Предположим, что вы хотите создать обобщенный класс, который содержит метод, возвращающий среднее значение массива чисел. Более того, вы хотите использовать этот класс для получения среднего значения чисел, включая целые числа и числа с плавающей точкой одинарной и двойной точности. То есть вы хотите указать тип числовых данных обобщенно, используя параметр типа. Чтобы создать такой класс, можно попробовать что-то вроде этого.

```
// Stats пытается (безуспешно) создать
// обобщенный класс, который вычисляет
// среднее значение массива чисел
// заданного типа.
//
// Класс содержит ошибку!
class Stats<T> {
    T[] nums; // nums — это массив элементов типа T

    // Передать конструктору ссылку
    // на массив значений типа T.
    Stats(T[] o) {
        nums = o;
    }

    // Возвращает double во всех случаях.
    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 типа `double` каждого числа в массиве `nums`, вызывая метод `doubleValue()`. Поскольку все числовые классы, такие как `Integer` и `Double`, являются подклассами `Number`, а класс `Number` определяет метод `doubleValue()`, этот метод доступен всем числовым классам-оболочкам. Проблема в том, что компилятор не имеет возможности узнать, что вы намерены создавать объекты класса `Stats`, используя только числовые типы. То есть, когда вы компилируете класс `Stats`, выдается сообщение об ошибке, свидетельствующее о том, что метод `doubleValue()` не известен. Чтобы решить эту проблему, вам нужен какой-то способ сообщить компилятору, что вы собираетесь передавать в параметре `T` только числовые типы. Более того, необходим еще некоторый способ *гарантии* того, что будут передаваться *только* числовые типы.

Чтобы справиться с этой ситуацией, язык Java предлагает *ограниченные типы*. Когда указывается параметр типа, вы можете создать ограничение сверху, которое объявляет суперкласс, от которого должны быть унаследованы все аргументы типов. Для этого используется ключевое слово `extends` при указании параметра типа, как показано ниже.

```
<T extends суперкласс>
```

Это означает, что параметр `T` может быть заменен только классом *суперкласс* либо его подклассами. То есть *суперкласс* объявляет включающую верхнюю границу. Вы можете использовать ограничение сверху, чтобы исправить класс `Stats`,

показанный выше, указав класс Number как верхнюю границу используемого параметра типа.

```
// В этой версии Stats аргумент типа
// T должен быть либо Number, либо классом,
// унаследованным от него.
class Stats<T extends Number> {
    T[] nums; // массив Number или подклассов

    // Передать конструктору ссылку на массив
    // элементов Number или его подклассов.
    Stats(T[] o) {
        nums = o;
    }

    // Возвратить double во всех случаях.
    double average() {
        double sum = 0.0;

        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue();

        return sum / nums.length;
    }
}

// Демонстрация 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);

        // Это не скомпилируется, потому что 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);
    }
}
```

Результат работы этой программы выглядит следующим образом.

```
Среднее значение iob равно 3.0
Среднее значение dob равно 3.3
```

Обратите внимание на то, что класс Stats теперь объявлен так.

```
class Stats<T extends Number> {
```

Поскольку тип T теперь ограничен классом Number, компилятор Java знает, что все объекты типа T могут вызывать метод doubleValue(), так как это метод класса Number. Это уже серьезное преимущество. Однако в качестве дополнительного бонуса ограничение параметра T также предотвращает создание нечисловых объектов класса Stats. Например, если вы попытаетесь убрать комментарии в строках,

находящихся в конце программы, и перекомпилировать ее, то получите ошибку времени компиляции, потому что класс `String` не является подклассом `Number`.

В дополнение к использованию типа класса как ограничения, вы можете также применять тип интерфейса. Фактически вы можете указывать в качестве ограничений множество интерфейсов. Более того, такое ограничение может включать как тип класса, так и один или более интерфейсов. В этом случае тип класса должен быть задан первым. Когда ограничение включает тип интерфейса, допустимы только аргументы типа, реализующие этот интерфейс. Указывая ограничение, имеющее класс и интерфейс либо множество интерфейсов, применяйте оператор `&` для их объединения.

```
class Gen<T extends MyClass & MyInterface> { // ...
```

Здесь параметр `T` ограничен классом по имени `MyClass` и интерфейсом `MyInteface`. То есть любой тип, переданный параметру `T`, должен быть подклассом класса `MyClass` и иметь реализацию интерфейса `MyInteface`.

Использование шаблонов аргументов

Кроме того, что контроль типов удобен сам по себе, иногда он позволяет получить чрезвычайно полезные конструкции. Например, класс `Stats`, рассмотренный в предыдущем разделе, предполагает, что вы хотите добавить метод по имени `sameAvg()`, который определяет, содержат ли два объекта класса `Stats` массивы, порождающие одинаковое среднее значение, независимо от того, какого типа числовые значения в них содержатся. Например, если один объект содержит значения типа `double` 1.0, 2.0 и 3.0, а другой — целочисленные значения 2, 1 и 3, то среднее значение у них будет одинаково. Один из способов реализации метода `sameAvg()` — передать ему аргумент класса `Stats`, а затем сравнивать его среднее значение со средним значением вызывающего объекта, возвращая значение `true`, если они равны. Например, необходимо иметь возможность вызывать метод `sameAvg()`, 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.sameAvg(dob))
    System.out.println("Средние значения одинаковы.");
else
    System.out.println("Средние значения отличаются.");
```

Вначале написание метода `sameAvg()` кажется простой задачей. Поскольку класс `Stats` является обобщенным и его метод `average()` может работать с объектами класса `Stats` любого типа, кажется, что написание метода `sameAvg()` не представляет сложности. К сожалению, проблема появляется сразу, как только вы попытаетесь объявить параметр типа `Stats`. Поскольку `Stats` — параметризованный тип, какой тип параметра вы укажете для `Stats`, когда создадите параметр типа `Stats`?

Сначала вы можете подумать о решении вроде такого, в котором параметр `T` будет использоваться как параметр типа.

```
// Это не сработает!
// Определение эквивалентности двух средних значений.
```

```
boolean sameAvg(Stats<T> ob) {
    if(average() == ob.average())
        return true;

    return false;
```

С приведенным выше кодом связана проблема, которая заключается в том, что такой код будет работать только с другим объектом класса `Stats`, тип которого совпадает с вызывающим объектом. Например, если вызывающий объект имеет тип `Stats<Integer>`, то параметр `ob` также должен иметь тип `Stats<Integer>`. Он, например, не может применяться для сравнения среднего значения типа `Stats<Double>` со средним значением типа `Stats<Short>`. Таким образом, этот подход будет работать только в очень ограниченном контексте и не представляет собой общего (а стало быть, и обобщенного) решения.

Чтобы создать обобщенную версию метода `sameAvg()`, следует использовать другое средство обобщений Java – шаблоны аргументов. Шаблон аргумента указывается символом `?` и представляет собой неизвестный тип. Применение шаблона – единственный способ написать работающий метод `sameAvg()`.

```
// Определение эквивалентности двух средних значений.
// Отметим использование шаблонного символа.
boolean sameAvg(Stats<?> ob) {
    if(average() == ob.average())
        return true;

    return false;
}
```

Здесь часть `Stats<?>` соответствует любому объекту класса `Stats`, что позволяет сравнивать между собой средние значения любых двух объектов класса `Stats`. Это демонстрируется в следующей программе.

```
// Использование шаблона.
class Stats<T extends Number> {
    T[] nums; // массив Number или подклассов

    // Передать конструктору ссылку на массив
    // типа Number или его подклассов.
    Stats(T[] o) {
        nums = o;
    }

    // Во всех случаях возвращает double.
    double average() {
        double sum = 0.0;

        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue();

        return sum / nums.length;
    }

    // Определение эквивалентности двух средних.
    // Обратите внимание на использование шаблонов.
    boolean sameAvg(Stats<?> ob) {
        if(average() == ob.average())
            return true;

        return false;
```

```
    }
}

// Демонстрация шаблона.
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 для 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 для 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 для fob равно " + x);

        // Посмотреть, какие массивы имеют одинаковые средние.
        System.out.print("Средние iob и dob ");
        if(iob.sameAvg(dob))
            System.out.println("равны.");
        else
            System.out.println("отличаются.");

        System.out.print("Средние iob и fob ");
        if(iob.sameAvg(fob))
            System.out.println("равны.");
        else
            System.out.println("отличаются.");
    }
}
```

Результат работы этой программы.

```
Среднее для iob равно 3.0
Среднее для dob равно 3.3
Среднее для fob равно is 3.0
Средние iob и dob отличаются.
Средние iob и fob равны.
```

И еще один, последний, момент: важно понимать, что шаблон не влияет на то, какого конкретного типа создается объект класса `Stats`. Этим управляет слово `extends` в объявлении класса `Stats`. Шаблон просто соответствует *корректному* объекту класса `Stats`.

Ограниченные шаблоны

Шаблоны аргументов могут быть ограничены почти таким же способом, как параметры типов. Ограниченный шаблон особенно важен, когда вы создаете обобщенный тип, оперирующий иерархией классов. Чтобы понять, почему, обратимся к примеру. Рассмотрим следующую иерархию классов, которые инкапсулируют координаты.

```
// Двухмерные 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 двухмерные координаты XY. Его наследник – класс `ThreeD` – добавляет третье измерение, описывая координаты XYZ. От класса `ThreeD` наследуется класс `FourD`, который добавляет четвертое измерение (время), порождая четырехмерные координаты.

Ниже показан обобщенный класс, называемый `Coords`, который хранит массив координат.

```
// Этот класс хранит массив координатных объектов.
class Coords<T extends TwoD> {
    T[] coords;

    Coords(T[] o) { coords = o; }
}
```

Обратите внимание на то, что класс `Coords` задает тип параметра, ограниченный классом `TwoD`. Это значит, что любой массив, сохраненный в объекте класса `Coords`, будет содержать объект типа `TwoD` или любой из его подклассов.

Теперь предположим, что вы хотите написать метод, который отображает координаты X и Y для каждого элемента в массиве `coords` объекта класса `Coords`. Поскольку все типы объектов класса `Coords` имеют, как минимум, пару координат (X и Y), это легко сделать с помощью шаблона.

```
static void showXY(Coords<?> c) {
    System.out.println("X Y Coordinates:");
    for(int i=0; i < c.coords.length; i++)
        System.out.println(c.coords[i].x + " " + c.coords[i].y);
    System.out.println();
}
```

Поскольку класс `Coords` – ограниченный обобщенный тип, который задает класс `TwoD` как верхнюю границу, все объекты, которые можно использовать для создания объекта класса `Coords`, будут массивами типа `TwoD` или классов, наследуемых от него. Таким образом, метод `showXY()` может отображать содержимое любого объекта класса `Coords`.

Но что если вы хотите создать метод, отображающий координаты X, Y и Z объекта классов ThreeD или FourD? Беда в том, что не все объекты класса Coords будут иметь три координаты, так как тип Coords<TwoD> будет иметь только координаты X и Y. Как же написать метод, который будет отображать координаты X, Y и Z для типов Coords<ThreeD> и Coords<FourD>, в то же время предотвращая использование этого метода с объектами класса Coords<TwoD>? Ответ заключается в использовании *ограниченных шаблонов аргументов.*

Ограниченный шаблон задает верхнюю или нижнюю границу типа аргумента. Это позволяет ограничить типы объектов, которыми будет оперировать метод. Наиболее популярен шаблон, ограничивающий сверху, который создается с применением оператора extends, почти так же как при описании ограниченного типа.

Применяя ограниченные шаблоны, легко создать метод, отображающий координаты X, Y и Z для объекта класса Coords, если этот объект действительно имеет эти три координаты. Например, следующий метод showXYZ() показывает координаты элементов, сохраненных в объекте класса Coords, если эти элементы имеют тип ThreeD (или унаследованы от класса ThreeD).

```
static void showXYZ(Coords<? extends ThreeD> c) {
    System.out.println("X Y Z Coordinates:");
    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 на то, что слово extends может быть добавлено к шаблону в параметре объявления параметра c. Таким образом, знакоместу ? должен соответствовать любой тип до тех пор, пока он является типом ThreeD или типом, унаследованным от него. То есть ключевое слово extends накладывает верхнее ограничение соответствия на знакоместо ?. Из-за этого ограничения метод showXYZ() может быть вызван со ссылкой на объекты типа Coords<ThreeD> или Coords<FourD>, но не со ссылкой на тип Coords<TwoD>. Попытка вызвать метод showXYZ() со ссылкой на тип Coords<TwoD> вызывает ошибку времени компиляции, что обеспечивает безопасность типов.

Ниже приведена полная программа, которая демонстрирует действие ограниченного шаблона аргумента.

```
// Ограниченные шаблоны аргумента.

// Двухмерные 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 координатных объектов.
class Coords<T extends TwoD> {
    T[] coords;

    Coords(T[] o) { coords = o; }
}

// Демонстрация ограниченных шаблонов.
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); // OK, это TwoD
```

```
    // showXYZ(tdlocs); // Ошибка, не ThreeD
    // showAll(tdlocs); // Ошибка, не FourD

    // Теперь создаем 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.");
       // Здесь все ОК.
       showXY(fdlocs);
       showXYZ(fdlocs);
       showAll(fdlocs);
   }
}
```

Результат работы этой программы выглядит следующим образом.

```
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 следующие закомментированные строки.

```
// showXYZ(tdlocs); // Ошибка, не ThreeD
// showAll(tdlocs); // Ошибка, не FourD
```

Поскольку `tdlocs` — это объект класса `Coords<TwoD>`, он не может быть использован для вызова методов `showXYZ()` или `showAll()`, потому что ограничивающий шаблон аргумента в их объявлении предотвращает это. Чтобы убедиться в этом, попробуйте убрать комментарии с упомянутых строк и попытаться скомпилировать программу. Вы получите ошибку компиляции по причине несоответствия типов.

В общем случае, для того чтобы установить верхнюю границу шаблона, используйте следующий тип шаблонного выражения.

```
<? extends суперкласс>
```

Здесь *суперкласс* — это имя класса, который служит верхней границей. Помните, что это — включающее выражение, потому что класс, заданный в качестве верхней границы (т.е. *суперкласс*), также находится в пределах допустимых типов.

Вы также можете указать нижнюю границу шаблона, добавив выражение `super` к объявлению шаблона. Вот его общая форма.

```
<? super подклассы>
```

В этом случае допустимыми аргументами могут быть только классы, которые являются суперклассами для *подклассы*. Это исключающая конструкция, поскольку она не включает класс *подклассы*.

Создание обобщенного метода

Как было показано в предыдущих примерах, методы внутри обобщенного класса могут использовать параметр типа, а следовательно, обобщения относятся также к параметрам методов. Однако можно объявить обобщенный метод, который сам по себе использует один или более параметров типа. Более того, можно объявить обобщенный метод, который включен в непараметризованный (необобщенный) класс.

Начнем с примера. В следующей программе объявлен необобщенный класс по имени `GenMethDemo` и статический обобщенный метод внутри класса по имени `isIn()`. Метод `isIn()` определяет, является ли объект членом массива. Он может быть использован с любым типом объектов и массивов до тех пор, пока массив содержит объекты, совместимые с типом искомого объекта.

```
// Демонстрация простого обобщенного метода.
class GenMethDemo {

    // Определение, содержится ли объект в массиве.
    static <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[]) {
        // Применение isIn() для Integer.
        Integer nums[] = { 1, 2, 3, 4, 5 };

        if(isIn(2, nums))
            System.out.println("2 содержится в nums");

        if(!isIn(7, nums))
            System.out.println("7 не содержится в nums");

        System.out.println();

        // Применение isIn() для String.
        String strs[] = { "один", "два", "три",
                          "четыре", "пять" };

        if(isIn("два", strs))
            System.out.println("два содержится в strs");

        if(!isIn("семь", strs))
            System.out.println("семь содержится в strs");
```

```
        // Не скомпилируется! Типы должны быть совместимыми.
        // if(isIn("два", nums))
        // System.out.println("два содержится в strs ");
    }
}
```

Результат работы этой программы показан ниже.

```
2 содержится в nums
7 не содержится в nums

два содержится в strs
семь не содержится в strs
```

Рассмотрим метод `isIn()` поближе. Для начала посмотрите, как объявлен метод в следующей строке.

```
static <T, V extends T> boolean isIn(T x, V[] y) {
```

Параметр типа объявлен перед типом возвращаемого значения метода. Тип `V` ограничен сверху типом `T`. То есть тип `V` либо должен быть тем же типом, что и `T`, либо типом его подклассов. Это отношение указывает, что метод `isIn()` может быть вызван только с аргументами, совместимыми между собой. Также обратите внимание на то, что метод `isIn()` статический, что позволяет вызывать его независимо от какого-либо объекта. Однако ясно, что обобщенные методы могут быть как статическими, так и нестатическими. Нет никаких ограничений на этот счет.

Теперь обратите внимание на то, как метод `isIn()` вызывается внутри метода `main()`, — с нормальным синтаксисом вызовов, без необходимости указывать аргументы типа. Дело в том, что типы аргументов подставляются автоматически, и типы `T` и `V` определяются соответственно. Например, в вызове

```
if(isIn(2, nums))
```

тип первого аргумента — `Integer` (благодаря автоупаковке), поэтому вместо параметра `T` подставляется класс `Integer`. Базовый тип второго аргумента — также класс `Integer`, что подставляет его и вместо параметра `V`. Во втором вызове используются типы `String`, и вместо параметров типа `T` и `V` подставляются классы `String`. Теперь обратите внимание на закомментированный код.

```
// if(isIn("два", nums))
//     System.out.println("два содержится в strs ");
```

Если вы уберете комментарии с этих строк, а затем попытаетесь скомпилировать программу, то получите ошибку. Причина в том, что тип параметра типа `V` ограничен типом параметра типа `T` в конструкции `extends` объявления параметра типа `V`. Это значит, что параметр `V` должен иметь либо тип `T`, либо тип его подкласса. А в этом случае первый аргумент имеет тип `String`, но второй — тип `Integer`, который не является подклассом класса `String`. Это вызовет ошибку несоответствия типов во время компиляции. Такая способность обеспечивать безопасность типов — одно из наиболее существенных преимуществ обобщенных методов.

Синтаксис, использованный для создания метода `isIn()`, можно обобщить. Вот синтаксис обобщенного метода.

```
<список_параметров_типа> ссылочный_тип имя_метода (список_параметров)
{ // ...
```

Во всех случаях *список_параметров_типа* — это разделенный запятыми список параметров типа. Обратите внимание на то, что для обобщенного метода список параметров типа предшествует типу возвращаемого значения.

Обобщенные конструкторы

Конструкторы также могут быть обобщенными, даже если их классы таковыми не являются. Например, рассмотрим следующую короткую программу.

```
// Использование обобщенного конструктора.
class GenCons {
    private double val;

    <T extends 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();
    }
}
```

Вывод этой программы.

```
val: 100.0
val: 123.5
```

Поскольку конструктор `GenCons()` задает параметр обобщенного типа, который может быть подклассом класса `Number`, конструктор `GenCons()` можно вызывать с любым числовым типом, включая классы `Integer`, `Float` или `Double`. Таким образом, даже несмотря на то, что класс `GenCons` — необобщенный, его конструктор обобщен.

Обобщенные интерфейсы

В дополнение к обобщенным классам и методам, вы можете объявлять обобщенные интерфейсы. Обобщенные интерфейсы указывают так же, как и обобщенные классы. Ниже показан пример. В нем создается интерфейс `MinMax`, объявляющий методы `min()` и `max()`, которые, как ожидается, должны возвращать минимальное и максимальное значения из некоторого множества объектов.

```
// Пример обобщенного интерфейса.

// Интерфейс Min/Max.
interface MinMax<T extends Comparable<T>> {
    T min();
    T max();
}

// Реализация MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T> {
    T[] vals;
```

```
    MyClass(T[] o) { vals = o; }

    // Возвращает минимальное значение из 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;
    }

    // Возвращает максимальное значение из 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());
    }
}
```

Результат работы этой программы.

```
Максимальное значение в inums: 8
Минимальное значение в inums: 2
Максимальное значение в chs: w
Минимальное значение в chs: b
```

Хотя большинство аспектов этой программы просты для понимания, некоторые ключевые моменты следует пояснить. В первую очередь обратите внимание на то, как объявлен интерфейс `MinMax`.

```
interface MinMax<T extends Comparable<T>> {
```

В общем случае обобщенный интерфейс объявляется так же, как и обобщенный класс. В этом случае параметр типа `T` имеет верхнюю границу — интерфейс `Comparable`, определенный в пакете `java.lang`. Класс, который реализует интерфейс `Comparable`, *определяет объекты, которые могут быть упорядочены*. То есть требование, чтобы параметр типа `T` был ограничен сверху интерфейсом `Comparable`, гарантирует, что интерфейс `MinMax` может быть использован толь-

ко с объектами, которые могут сравниваться между собой. (Более подробную информацию об интерфейсе Comparable можно найти в главе 16.) Отметим, что интерфейс Comparable сам по себе также является обобщенным. Он принимает параметр типа объектов, которые должны сравниваться.

Далее класс MyClass реализует интерфейс MinMax. Вот как выглядит объявление класса MyClass.

```
class MyClass<T extends Comparable<T>> implements MinMax<T> {
```

Обратите особое внимание на способ, которым параметр типа T объявлен в классе MyClass, а затем передан интерфейсу MinMax. Поскольку интерфейс MinMax требует типа, который реализует интерфейс Comparable, реализующий класс (в данном случае — интерфейс MinMax) должен указать то же ограничение. Более того, однажды установленное это ограничение уже не нужно повторять в операторе implements. Фактически так поступать некорректно. Например, следующая строка неверна и не может быть скомпилирована.

```
// Это не правильно!
class MyClass<T extends Comparable<T>>
    implements MinMax<T extends Comparable<T>> {
```

Однажды установленный параметр типа просто передается интерфейсу без последующих модификаций.

Вообще, если класс реализует обобщенный интерфейс, то такой класс также должен быть обобщенным, по крайней мере, в тех пределах, где он принимает параметр типа, переданный интерфейсу. Например, следующая попытка объявления класса MyClass вызовет ошибку.

```
class MyClass implements MinMax<T> { // Неверно!
```

Поскольку класс MyClass не объявляет параметра, нет способа передать его интерфейсу MinMax. В этом случае идентификатор параметра T просто неизвестен, и компилятор выдаст ошибку. Конечно, если, как показано ниже, класс реализует *специфический тип* обобщенного интерфейса

```
class MyClass implements MinMax<Integer> { // OK
```

то реализующий класс не обязан быть обобщенным.

Обобщенный интерфейс представляет два преимущества. Во-первых, он может быть реализован для разных типов данных. Во-вторых, он позволяет включить ограничения на типы данных, для которых он может быть реализован. В примере интерфейса MinMax вместо параметра T могут быть подставлены только типы, совместимые с интерфейсом Comparable.

Вот общий синтаксис обобщенного интерфейса.

```
interface имя_интерфейса<список_параметров_типа> { // ...
```

Здесь *список_параметров_типа* — разделенный запятыми список параметров типа. Когда реализуется обобщенный интерфейс, следует указать аргументы типов, как показано ниже.

```
class имя_класса<список_параметров_типа>
    implements имя_интерфейса<список_аргументов_типов> {
```

Базовые типы и унаследованный код

Поскольку поддержка обобщений не существовала до JDK 5, необходим некоторый способ трансляции старого кода, разработанного до появления обобщений.

На момент написания этой книги существует большое количество старого кода, который должен оставаться функциональным и быть совместимым с обобщениями. Код, предшествующий обобщениям, должен иметь возможность работать с обобщенным кодом, и обобщенный код должен работать со старым кодом.

Чтобы облегчить переход к обобщениям, язык Java позволяет обобщенным классам быть использованными без аргументов типа. Это создает *базовый тип* ("сырой" тип) для класса. Этот базовый тип совместим с унаследованным кодом, который не имеет представления об обобщенном синтаксисе. Главный недостаток использования базового типа в том, что безопасность типов утрачивается.

Вот пример, демонстрирующий базовый тип в действии.

```
// Демонстрация базового типа.
class Gen<T> {

    T ob; // Объявление объекта типа T

    // Передача конструктору ссылки
    // на объект типа T.
    Gen(T o) {
        ob = o;
    }

    // Возврат ob.
    T getob() {
        return ob;
    }
}

// Демонстрация базового типа.
class RawDemo {
    public static void main(String args[]) {

        // Создать объект Gen для Integer.
        Gen<Integer> iOb = new Gen<Integer>(88);

        // Создать объект Gen для String.
        Gen<String> strOb = new Gen<String>("Обобщенный тест");

        // Создать объект базового типа Gen и дать ему значение Double.
        Gen raw = new Gen(new Double(98.6));

        // Приведение необходимо, поскольку тип неизвестен.
        double d = (Double) raw.getob();
        System.out.println("значение: " + d);

        // Использование базовых типов может вызвать исключения
        // времени выполнения. Ниже представлены некоторые примеры.
        // Следующее приведение вызовет ошибку времени выполнения!
        // int i = (Integer) raw.getob(); // ошибка времени выполнения

        // Это присваивание нарушает безопасность типов.
        strOb = raw; // OK, но потенциально неверно
        // String str = strOb.getob(); // ошибка времени выполнения

        // Это присваивание также нарушает безопасность типов.
        raw = iOb; // OK, но потенциально неверно
        // d = (Double) raw.getob(); // ошибка времени выполнения
    }
}
```

С этой программой связано несколько интересных моментов. Следующее объявление создает класс Gen базового типа.

```
Gen raw = new Gen(new Double(98.6));
```

Обратите внимание на то, что никаких аргументов типа не указывается. По сути, оператор создает объект класса Gen, тип T которого заменяется на тип Object.

Базовые типы не обеспечивают безопасности. То есть переменной базового типа можно присвоить ссылку на любой тип объектов класса Gen. Обратное также возможно: переменной специфического типа Gen можно присвоить ссылку на объект базового типа Gen. Однако обе операции потенциально небезопасны, так как механизм проверки типов при этом не применяется.

Недостаток безопасности иллюстрируется закомментированными строками в конце программы. Рассмотрим каждую из них.

```
// int i = (Integer) raw.getob(); // ошибка времени выполнения
```

В этом операторе получается значение объекта ob внутри объекта raw, и это значение приводится к типу Integer. Проблема в том, что объект raw содержит значение класса Double вместо целого числа. Однако это не может быть обнаружено на этапе компиляции, поскольку тип объекта raw неизвестен. То есть этот оператор вызовет сбой во время выполнения.

Следующая последовательность кода присваивает объект strOb (ссылка на тип Gen<String>) ссылке на объект класса Gen.

```
strOb = raw; // OK, но потенциально неверно
// String str = strOb.getob(); // ошибка времени выполнения
```

Это присваивание само по себе синтаксически корректно, но сомнительно. Поскольку объект strOb имеет тип Gen<String>, предполагается, что он содержит строку. Однако после присваивания объект, на который ссылается объект strOb, содержит значение типа Double. То есть во время выполнения, когда предпринимается попытка присвоить объект strOb переменной str, происходит ошибка времени выполнения, так как объект strOb теперь содержит значение типа Double. То есть присваивание базовой ссылки обобщенной ссылке минует механизм проверки типов.

Следующая последовательность представляет собой противоположный случай.

```
raw = iOb; // OK, но потенциально неверно
// d = (Double) raw.getob(); // ошибка времени выполнения
```

Здесь обобщенная ссылка присваивается переменной базовой ссылки. Хотя это синтаксически корректно, но также может привести к проблемам, как показывает вторая строка. В этом случае объект raw теперь ссылается на объект, содержащий значение типа Integer, но приведение предполагает, что в нем содержится значение типа Double. Эта ошибка не может быть предотвращена во время компиляции. Вместо этого она проявляется во время выполнения.

Из-за того, что базовые типы представляют опасность, javac отображает *непроверенные предупреждения*, когда обнаруживает, что их использование может нарушить безопасность типов. В предыдущей программе следующие строки вызовут появление таких предупреждений.

```
Gen raw = new Gen(new Double(98.6));

strOb = raw; // OK, но потенциально неверно
```

В первой строке происходит вызов конструктора Gen() без аргумента типа, что вызывает предупреждение. Во второй строке выполняется присваивание ба-

зовой ссылки обобщенной переменной, что также вызывает появление предупреждения.

На первый взгляд может показаться, что эта строка также должна порождать предупреждение, но этого не происходит.

```
raw = iOb; // OK, но потенциально неверно
```

Здесь не выдается никаких предупреждений компилятора, потому что присваивание не вызывает никакой *дополнительной* потери безопасности типов, кроме той, что уже происходит при создании объекта `raw`.

Один заключительный момент: следует ограничивать использование базовых типов теми случаями, когда приходится смешивать унаследованный код с новым, обобщенным. Базовые типы — это просто средство переноса, а не что-то такое, что должно применяться в новом коде.

Иерархии обобщенных классов

Обобщенные классы могут быть частью иерархии классов — так же, как и любые другие необобщенные классы. То есть обобщенный класс может выступать в качестве суперкласса или подкласса. Ключевое отличие между обобщенными и необобщенными иерархиями состоит в том, что в обобщенной иерархии любые аргументы типов, необходимые обобщенному суперклассу, всеми подклассами должны передаваться по иерархии вверх. Это похоже на способ, которым аргументы конструкторов передаются по иерархии.

Использование обобщенного суперкласса

Ниже приведен пример иерархии, которая использует обобщенный суперкласс.

```
// Простая иерархия обобщенных классов.
class Gen<T> {
    T ob;

    Gen(T o) {
        ob = o;
    }

    // Возвращает ob.
    T getob() {
        return ob;
    }
}

// Подкласс Gen.
class Gen2<T> extends Gen<T> {
    Gen2(T o) {
        super(o);
    }
}
```

В этой иерархии класс `Gen2` расширяет обобщенный класс `Gen`. Обратите внимание на то, как в следующей строке объявлен класс `Gen2`.

```
class Gen2<T> extends Gen<T> {
```

Параметр типа T указан в объявлении класса Gen2 и также передается классу Gen в операторе extends. Это значит, что тип, переданный классу Gen2, будет также передан классу Gen. Например, объявление

```
Gen2<Integer> num = new Gen2<Integer>(100);
```

передает тип Integer как параметр типа классу Gen. То есть объект ob внутри части Gen класса Gen2 будет иметь тип Integer. Отметим также, что класс Gen2 никак не использует параметр типа T, кроме того, что передает его суперклассу Gen. Даже если подкласс обобщенного суперкласса никак не нуждается в том, чтобы быть обобщенным, все же он должен указывать параметры типа, необходимые его обобщенному суперклассу.

Конечно, при необходимости подкласс может добавлять свои собственные параметры типа. Например, ниже показан вариант предыдущей иерархии, в котором класс Gen2 добавляет собственный параметр типа.

```
// Подкласс может добавлять собственные параметры типа.
class Gen<T> {
    T ob; // Объявление объекта типа T

    // Передача конструктору
    // ссылки на объект типа T.
    Gen(T o) {
        ob = o;
    }

    // Возвращает ob.
    T getob() {
        return ob;
        }
    }

    // Подкласс Gen, который определяет
    // 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;
        }
    }
    // Создание объекта типа Gen2.
    class HierDemo {
        public static void main(String args[]) {

        // Создание объектов Gen2 для String и Integer.
        Gen2<String, Integer> x =
                new Gen2<String, Integer>("Значение равно: ", 99);

        System.out.print(x.getob());
        System.out.println(x.getob2());
    }
}
```

Обратите внимание на объявление класса Gen2, показанное в следующей строке.

```
class Gen2<T, V> extends Gen<T> {
```

Здесь T — тип, переданный классу Gen, а V — тип, специфичный для класса Gen2. Параметр типа V используется при объявлении объекта, названного ob2, а также в качестве типа возвращаемого значения метода getob2(). В методе main() создается объект класса Gen2 типа String для параметра T и типа Integer для параметра V.

Программа выдает следующий вполне ожидаемый результат.

```
Значение равно: 99
```

Обобщенный подкласс

Суперклассом для обобщенного класса вполне может выступать необобщенный класс. Например, рассмотрим программу.

```
// Необобщенный класс может быть суперклассом
// для 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; // Объявление объекта типа T

    // Передать конструктору объект
    // типа T.
    Gen(T o, int i) {
        super(i);
        ob = o;
    }

    // Возвращает 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());
    }
}
```

Результат работы этой программы показан ниже.

```
Добро пожаловать 47
```

Обратите внимание на то, как в этой программе класс Gen наследуется от класса NonGen.

```
class Gen<T> extends NonGen {
```

Поскольку класс NonGen — необобщенный, никакие аргументы типа не указываются. То есть даже если класс Gen объявляет параметр типа T, он не требуется (и не может быть использован) классом NonGen. То есть класс Gen наследуется от класса NonGen обычным способом. Никаких специальных условий не требуется.

Сравнение типов обобщенной иерархии во время выполнения

Вспомните оператор получения информации о типе времени выполнения — instanceof, который был описан в главе 13. Как уже объяснялось, оператор instanceof определяет, является ли объект экземпляром класса. Он возвращает значение true, если объект относится к указанному типу либо может быть приведен к этому типу. Оператор instanceof может быть применим к объектам обобщенных классов. Следующий класс демонстрирует следствия совместимости типов в обобщенных иерархиях.

```
// Использование оператора instanceof с иерархией обобщенных классов.
class Gen<T> {
    T ob;

    Gen(T o) {
        ob = o;
    }

    // Возвращает ob.
    T getob() {
        return ob;
    }
}

// Подкласс Gen.
class Gen2<T> extends Gen<T> {
    Gen2(T o) {
        super(o);
    }
}

// Демонстрация определения идентификатора
// типа в 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>("Обобщенный тест");

        // Проверить, является ли iOb2 какой-то из форм Gen2.
        if(iOb2 instanceof Gen2<?>)
            System.out.println("iOb2 является экземпляром Gen2");
```

```
        // Проверить, является ли iOb2 какой-то из форм Gen.
        if(iOb2 instanceof Gen<?>)
            System.out.println("iOb2 является экземпляром Gen");
        System.out.println();

        // Проверить, является ли strOb2 объектом Gen2.
        if(strOb2 instanceof Gen2<?>)
            System.out.println("strOb2 является экземпляром Gen2");

        // Проверить, является ли strOb2 объектом Gen.
        if(strOb2 instanceof Gen<?>)
            System.out.println("strOb2 является экземпляром Gen");

        System.out.println();

        // Проверить, является ли iOb экземпляром Gen2, что не так.
        if(iOb instanceof Gen2<?>)
            System.out.println("iOb является экземпляром Gen2");

        // Проверить, является ли iOb экземпляром Gen, что так и есть.
        if(iOb instanceof Gen<?>)
            System.out.println("iOb является экземпляром Gen");

        // Следующее не скомпилируется, потому что информация
        // об обобщенном типе во время выполнения отсутствует.
        // if(iOb2 instanceof Gen2<Integer>)
        // System.out.println("iOb2 является экземпляром Gen2<Integer>");
    }
}
```

Вывод программы показан здесь.

```
iOb2 является экземпляром Gen2
iOb2 является экземпляром Gen

strOb2 является экземпляром Gen2
strOb2 является экземпляром Gen

iOb является экземпляром Gen
```

В этой программе класс `Gen2` — подкласс класса `Gen`, который обобщен по типу параметра `T`. В методе `main()` создается три объекта. Первый, `iOb`, — объект типа `Gen<Integer>`. Второй, `iOb2`, — объект типа `Gen2<Integer>`. И наконец, третий, `strOb2`, — объект типа `Gen2<String>`.

Затем программа использует оператор `instanceof` для проверки типа `iOb2`:

```
// Проверить, является ли iOb2 какой-то из форм Gen2.
if(iOb2 instanceof Gen2<?>)
    System.out.println("iOb2 является экземпляром Gen2");

// Проверить, является ли iOb2 какой-то из форм Gen.
if(iOb2 instanceof Gen<?>)
    System.out.println("iOb2 является экземпляром Gen");
```

Как показывает вывод, обе проверки успешны. В первом случае объект `iOb2` проверяется на соответствие типу `Gen2<?>`. Эта проверка успешна, поскольку она просто подтверждает, что объект `iOb2` является объектом какого-то из типов `Gen2`. Применение шаблонного символа позволяет оператору `instanceof` определить, относится ли объект `iOb2` к какому-то из типов `Gen2`. Далее объект `iOb` проверяется на принадлежность к типу суперкласса `Gen<?>`. Это также верно, поскольку объект `iOb2` представляет собой некоторую форму суперкласса `Gen`.

Следующие несколько строк в методе `main()` показывают ту же последовательность (и некоторый результат) для объекта `strOb2`.

Далее объект `iOb`, являющийся экземпляром суперкласса `Gen<Integer>`, проверяется в следующих строках.

```
// Проверить, является ли iOb экземпляром Gen2, что не так.
if(iOb instanceof Gen2<?>)
    System.out.println("iOb является экземпляром Gen2");

// Проверить, является ли iOb экземпляром Gen, что так и есть.
if(iOb instanceof Gen<?>)
    System.out.println("iOb является экземпляром Gen");
```

Первый оператор `if` возвращает значение `false`, поскольку объект `iOb` не является никакой из форм типа `Gen2`. Следующая проверка успешна, потому что объект `iOb` является некоторым типом объекта класса `Gen`.

Теперь посмотрим внимательно на закомментированные строки.

```
// Следующее не скомпилируется, потому что информация
// об обобщенном типе отсутствует во время выполнения.
// if(iOb2 instanceof Gen2<Integer>)
//     System.out.println("iOb2 является экземпляром Gen2<Integer>");
```

Как следует из текста комментария, эти строки не компилируются, потому что они пытаются сравнить объект `iOb2` со специфическим типом `Gen2` — в данном случае с типом `Gen2<Integer>`. Помните, что во время выполнения не существует никакой доступной информации о типе. Таким образом, у оператора `instanceof` нет способа узнать, является ли объект `iOb2` экземпляром типа `Gen2<Integer>` или нет.

Приведение

Вы можете приводить один экземпляр обобщенного класса к другому, только если они между собой совместимы и их аргументы типов одинаковы. Например, для предыдущей программы следующее приведение корректно

```
(Gen<Integer>) iOb2 // допустимо
```

потому что объект `iOb2` является экземпляром типа `Gen<Integer>`. Однако следующее приведение

```
(Gen<Long>) iOb2 // недопустимо
```

недопустимо, поскольку объект `iOb2` не является экземпляром типа `Gen<Long>`.

Переопределение методов в обобщенном классе

Метод обобщенного класса, как и любой другой метод, может быть переопределен. Например, рассмотрим следующую программу, в которой переопределяется метод `getob()`.

```
// Переопределение обобщенного метода в обобщенном классе.
class Gen<T> {
    T ob; // Объявление объекта типа T

    // Передать конструктору ссылку
    // на объект типа T.
    Gen(T o) {
        ob = o;
    }
```

```
    // Возвращение ob.
    T getob() {
        System.out.print("getob() класса Gen: " );
        return ob;
    }
}
// Подкласс Gen, переопределяющий getob().
class Gen2<T> extends Gen<T> {

    Gen2(T o) {
        super(o);
    }

    // Переопределение getob().
    T getob() {
        System.out.print("getob() класса Gen2: ");
        return ob;
    }
}

// Демонстрация переопределения обобщенных методов.
class OverrideDemo {
    public static void main(String args[]) {

        // Создание объекта Gen для Integer.
        Gen<Integer> iOb = new Gen<Integer> (88);

        // Создание объекта Gen2 для Integer.
        Gen2<Integer> iOb2 = new Gen2<Integer> (99);

        // Создание объекта Gen2 для String.
        Gen2<String> strOb2 = new Gen2<String> ("Обобщенный тест");
        System.out.println(iOb.getob());
        System.out.println(iOb2.getob());
        System.out.println(strOb2.getob());
    }
}
```

Результат работы этой программы показан ниже.

```
getob() класса Gen: 88
getob() класса Gen2: 99
getob() класса Gen2: Обобщенный тест
```

Как подтверждает вывод, переопределенная версия метода `getob()` вызывается для объекта класса `Gen2`, но для объектов типа `Gen` вызывается версия суперкласса.

Выведение типов и обобщения

Начиная с JDK 7 можно использовать сокращенный синтаксис для создания экземпляра обобщенного типа. Для начала рассмотрим следующий обобщенный класс.

```
class MyClass<T, V> {
    T ob1;
    V ob2;

    MyClass(T o1, V o2) {
        ob1 = o1;
```

```
        ob2 = o2;
    }
    // ...
}
```

До JDK 7, чтобы создать экземпляр класса `MyClass`, вам следовало бы использовать оператор, подобный следующему.

```
MyClass<Integer, String> mcOb =
    new MyClass<Integer, String>(98, "A String");
```

Здесь аргументы типа (которыми являются типы `Integer` и `String`) определяются дважды: когда объявляется объект `mcOb` и когда экземпляр класса `MyClass` создается при помощи оператора `new`. Со времени появления обобщений в JDK 5, эта форма была обязательна для всех версий языка Java до JDK 7. Хотя здесь нет ничего неправильного, по существу, эта форма немного более избыточна, чем это могло бы быть. В директиве `new` тип аргументов типа может быть без проблем выведен из типа объекта `mcOb`, поэтому в действительности нет никаких причин определять их во второй раз. Чтобы разрешить эту ситуацию, в комплект JDK 7 добавлен синтаксический элемент, позволяющий избегать повторной спецификации.

В JDK 7 приведенное выше объявление может быть переписано так.

```
MyClass<Integer, String> mcOb = new MyClass<>(98, "A String");
```

Обратите внимание на то, что код создания экземпляра просто использует угловые скобки <>, являющиеся пустым списком аргументов типа. Они указывают компилятору вывести аргументы типа, необходимые конструктору, в операторе `new`. Основное преимущество синтаксиса выведения типов в том, что он короче и иногда существенно сокращает весьма длинные операторы объявления.

Обобщим описанное выше. Когда используется выведение типов, синтаксис объявления для обобщенной ссылки и создания экземпляра имеет такую общую форму.

```
имя_класса<список_аргументов_типа> имя_переменной =
new имя_класса <>(список_аргументов_конструктора);
```

Здесь список аргументов типа конструктора в операторе `new` пуст.

Выведение типов может быть также применено к передаче параметров. Например, если в класс `MyClass` добавляется метод

```
boolean isSame(MyClass<T, V> o) {
    if(ob1 == o.ob1 && ob2 == o.ob2) return true;
    else return false;
}
```

то следующий вызов в JDK 7 является вполне корректным.

```
if(mcOb.isSame(new MyClass<>(1, "test"))) System.out.println("Same");
```

В данном случае аргументы типа для аргумента, переданного методу `isSame()`, могут быть выведены.

Важно понять, что выведение типов не будет работать во всех случаях. Например, с учетом следующей иерархии классов

```
class A<T, V> {}
class B<T, V> extends A<T, V>{ }
```

приведенное ниже объявление (которое не использует выведение типов) вполне корректно.

```
MyClass<A<Integer, Long>, String> mcOb2 =
    new MyClass<A<Integer, Long>, String>(new B<Integer,
                                          Long>(), "Обобщения");
```

Здесь, поскольку базовая ссылка на класс может ссылаться на объект производного класса, для объекта mcOb2 вполне допустимо ссылаться на объект класса MyClass, имеющий тип

```
MyClass<B<Integer, Long>, String>
```

даже притом, что ссылка имеет следующий тип.

```
MyClass<A<Integer, Long>, String>
```

Однако попытка использовать выведение типов для сокращения строки, как показано далее, не сработает.

```
// Не будет работать!
MyClass<A<Integer, Long>, String> mcOb2 =
    new MyClass<>(new B<Integer, Long>(), "Обобщения");
```

В данном случае будет получено сообщение об ошибке несоответствия типов.

Поскольку синтаксис выведения типов – нововведение JDK 7 и не будет работать с устаревшими компиляторами, в примерах этой книги будет использоваться полный синтаксис объявления экземпляров обобщенных классов. Таким образом, примеры будут работать с любым компилятором Java, который поддерживает обобщения. Использование полного синтаксиса позволяет также яснее понять, что создается, а это очень важно для кода примеров, представленных в книге. Однако в собственном коде использование синтаксиса выведения типов упростит ваши объявления.

Очистка

Обычно детали того, как компилятор Java трансформирует исходный текст в объектный код, знать не нужно. Однако в случае с обобщениями некоторое общее представление о процессе важно, поскольку оно объясняет, как работает механизм обобщения и почему иногда его поведение несколько необычно. По этой причине мы приведем краткое описание того, как обобщения реализованы в языке Java.

Важное ограничение, которое было наложено на способ реализации обобщений в языке Java, заключалось в том, что необходимо было обеспечить совместимость с предыдущими версиями языка Java. Только представьте, что обобщенный код должен быть совместим со старым, не обобщенным, кодом. То есть любые изменения в синтаксисе языка Java либо в JVM должны были избегать нарушения старого кода. Способ, которым Java реализует обобщения для удовлетворения этому требованию, называется *очисткой* (erasure).

Рассмотрим в общих чертах, как она работает. При компиляции вашего кода Java вся информация об обобщенных типах удаляется (чистится). Это означает замену параметров типа их ограничивающим типом, которым является тип Object, если только никакого явного ограничения не указано, с последующим использованием необходимых приведений (как того требуют аргументы типа) для поддержки совместимости типов с типами, указанными в аргументах. Компилятор также обеспечивает эту совместимость типов. Такой подход к обобщениям означает, что никакой информации о типах во время выполнения не существует. Это просто механизм автоматической обработки исходного кода.

Чтобы лучше понять, как работает очистка, рассмотрим два следующих класса.

```
// Здесь T ограничен типом Object по умолчанию.
class Gen<T> {
    T ob; // Здесь T будет заменен Object
```

```
    Gen(T o) {
        ob = o;
    }

    // Возвращает ob.
    T getob() {
        return ob;
    }
}

// Здесь T ограничен String.
class GenStr<T extends String> {
    T str; // Здесь T будет заменен String

    GenStr(T o) {
        str = o;
    }

    T getstr() { return str; }
}
```

После того как эти классы компилируются, параметр типа `T` в классе `Gen` будет заменен типом `Object`. Параметр типа `T` в классе `GenStr` будет заменен типом `String`. Внутри кода классов `Gen` и `GenStr` для обеспечения корректной типизации применяется приведение. Например, следующая последовательность

```
Gen<Integer> iOb = new Gen<Integer>(99);

int x = iOb.getob();
```

будет скомпилирована, как если бы она была написана так.

```
Gen iOb = new Gen(99);

int x = (Integer) iOb.getob();
```

Благодаря очистке, кое-что работает несколько иначе, чем может показаться. Например, рассмотрим короткую программу, создающую два объекта обобщенного класса `Gen`.

```
class GenTypeDemo {
    public static void main(String args[]) {
        Gen<Integer> iOb = new Gen<Integer>(99);
        Gen<Float> fOb = new Gen<Float>(102.2F);

        System.out.println(iOb.getClass().getName());
        System.out.println(fOb.getClass().getName());
    }
}
```

Вывод этой программы показан ниже.

```
Gen
Gen
```

Как видите, типом объектов `iOb` и `fOb` является тип `Gen`, а не `Gen<Integer>` и `Gen<Float>`, как вы могли ожидать. Помните, что все параметры типов во время компиляции удаляются. Во время выполнения существуют только базовые типы.

Методы-мосты

Иногда компилятору приходится добавлять к классу так называемый *метод-мост* (bridge method), чтобы справиться с ситуациями, когда результат очистки типов в перегруженном методе подкласса не совпадает с тем, что получается при очистке в суперклассе. В этом случае создается метод, который использует очистку типов суперкласса, и этот метод вызывает соответствующий метод подкласса, выполняющий очистку. Конечно, методы-мосты появляются только на уровне кода виртуальной машины, они невидимы для вас и недоступны для непосредственного вызова.

Несмотря на то что методы-мосты — это не то, в чем вы обычно нуждаетесь и с чем имеете дело, все же полезно рассмотреть ситуацию, в которой они создаются. Взгляните на следующую программу.

```
// Ситуация, в которой создается метод-мост.
class Gen<T> {
    T ob; // объявить объект типа T

    // Передать конструктору ссылку на
    // объект типа T.
    Gen(T o) {
        ob = o;
    }

    // Возвращает ob.
    T getob() {
        return ob;
    }
}

// Подкласс Gen.
class Gen2 extends Gen<String> {

    Gen2(String o) {
        super(o);
    }

    // Ориентированная на строки перегрузка getob().
    String getob() {
        System.out.print("Вызван String getob(): ");
        return ob;
    }
}

// Демонстрация ситуации, в которой необходим метод-мост.
class BridgeDemo {
    public static void main(String args[]) {

        // Создать объект Gen2 для Strings.
        Gen2 strOb2 = new Gen2("Обобщенный тест");

        System.out.println(strOb2.getob());
    }
}
```

В этой программе класс `Gen2` расширяет класс `Gen`, но делает это с использованием специфичной строковой версии класса `Gen`, как показывает следующее объявление.

```
class Gen2 extends Gen<String> {
```

Более того, внутри класса Gen2 метод getob() переопределен с типом возвращаемого значения String.

```
// Ориентированная на строки перегрузка getob().
String getob() {
    System.out.print("Вызван String getob(): ");
    return ob;
}
```

Все это совершенно допустимо. Единственная проблема в том, что из-за очистки типов ожидаемая форма метода getob() будет выглядеть так.

```
Object getob() { // ...
```

Чтобы справиться с этой проблемой, компилятор создает метод-мост с показанной выше сигнатурой, который вызывает строковую версию. То есть, если вы посмотрите на интерфейс класса Gen2 с помощью javap, то увидите следующие методы.

```
class Gen2 extends Gen<java.lang.String> {
    Gen2(java.lang.String);
    java.lang.String getob();
    java.lang.Object getob(); // метод-мост
}
```

Как видите, сюда включен метод-мост. (Комментарий добавлен автором, а не javap, а вывод, который вы видите, может измениться в зависимости от используемой версии языка Java.)

Последнее замечание о методах-мостах. Обратите внимание на то, что единственная разница между двумя методами getob() заключается в типе возвращаемого значения. Обычно это вызывает ошибку, но поскольку происходит не в исходном коде, проблема не возникает и JVM успешно справляется с этой ситуацией.

Ошибки неоднозначности

Включение в язык обобщений породило новый тип ошибок, от которых вам нужно защищаться, — *неоднозначность* (ambiquity). Ошибки неоднозначности случаются, когда очистка порождает два внешне разных обобщенных объявления, разрешаемых в виде одного очищенного типа, что вызывает конфликт. Вот пример, который включает перегрузку методов.

```
// Неоднозначность порождается очисткой перегруженных методов.
class MyGenClass<T, V> {
    T ob1;
    V ob2;

    // ...

    // Эти два перегруженных метода неоднозначны
    // и не компилируются.
    void set(T o) {
        ob1 = o;
    }

    void set(V o) {
        ob2 = o;
    }
}
```

Обратите внимание на то, что класс MyGenClass объявляет два обобщенных типа — T и V. Внутри класса MyGenClass предпринимается попытка перегрузить метод set() на основе параметров T и V. Это выглядит вполне резонно, потому что кажется, что параметры типа T и V содержат разные типы. Однако здесь возникают две проблемы неоднозначности.

Первая — судя по тому, как написан класс MyGenClass, нет требования, чтобы параметры T и V содержали разные типы. Например, объект MyGenClass, в принципе, вполне можно создать следующим образом.

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

В этом случае параметры типа T и V будут заменены типом String. Это делает обе версии метода set() идентичными, что, конечно же, представляет собой ошибку.

Вторая, более фундаментальная проблема состоит в том, что очистка типов приводит обе версии метода к следующему виду.

```
void set(Object o) { // ...
```

То есть перегрузка метода set(), которую пытается осуществить класс MyGenClass, в действительности неоднозначна.

Ошибки неоднозначности иногда трудно исправить. Например, если вы знаете, что параметр типа V всегда будет получать некий подтип класса String, то можете попытаться исправить класс MyGenClass, переписав его объявление следующим образом.

```
class MyGenClass<T, V extends String> { // почти OK!
```

Это позволит скомпилировать класс MyGenClass, и вы даже сможете создавать экземпляры его объектов примерно так.

```
MyGenClass<Integer, String> x = new MyGenClass<Integer, String>();
```

Это работает, потому что Java может аккуратно определить, когда какой метод должен быть вызван. Однако неоднозначность возникнет, когда вы попытаетесь выполнить следующую строку.

```
MyGenClass<String, String> x = new MyGenClass<String, String>();
```

Поскольку в этом случае параметры типа T и V получают тип String, то какую версию метода set() нужно вызвать? Вызов метода set() теперь неоднозначен.

Честно говоря, в предыдущем примере было бы лучше использовать два метода с разными именами, вместо того чтобы перегружать метод set(). Разрешение неоднозначности зачастую требует реструктуризации кода, потому что неоднозначность свидетельствует о концептуальной ошибке в дизайне.

Некоторые ограничения обобщений

Существует несколько ограничений, о которых следует помнить, применяя обобщения. Они включают создание объектов типа параметров, статических членов, исключений и массивов. Каждое из них мы рассмотрим далее.

Нельзя создавать экземпляр типа параметра

Создать экземпляр типа параметра невозможно. Например, рассмотрим такой класс.

```
// Нельзя создавать экземпляр типа T.
class Gen<T> {
    T ob;

    Gen() {
        ob = new T(); // Недопустимо!!!
    }
}
```

Здесь осуществляется недопустимая попытка создать экземпляр типа T. Причину понять просто: поскольку тип T не существует во время выполнения, как компилятор может узнать, какого типа объект следует создать? Вспомните, что очистка удаляет все параметры типа в процессе компиляции.

Ограничения на статические члены

Никакой статический член не может использовать тип параметра, объявленный в его классе. Например, оба статических члена этого класса являются недопустимыми.

```
class Wrong<T> {
    // Неверно, нельзя создать статические переменные типа T.
    static T ob;

    // Неверно, ни один статический метод не может использовать T.
    static T getob() {
        return ob;
    }
}
```

Несмотря на то что вы не можете объявить статические члены, которые используют тип параметра, объявленный в окружающем классе, вы *можете* объявлять обобщенные статические методы, определяющие их собственные параметры типа, как это делалось ранее в настоящей главе.

Ограничения обобщенных массивов

Существует два важных ограничения обобщений, касающиеся массивов. Во-первых, вы не можете создать экземпляр массива, тип элемента которого — параметр типа. Во-вторых, вы не можете создать массив специфичных для типа обобщенных ссылок. В следующей короткой программе демонстрируются обе ситуации.

```
// Обобщения и массивы.
class Gen<T extends Number> {
    T ob;

    T vals[]; // OK

    Gen(T o, T[] nums) {
        ob = o;

        // Этот оператор неверен.
        // vals = new T[10]; // нельзя создавать массив объектов T

        // Однако этот оператор верен.
        vals = nums; // можно присвоить ссылку существующему массиву
    }
}
```

```
class GenArrays {
    public static void main(String args[]) {
        Integer n[] = { 1, 2, 3, 4, 5 };

        Gen<Integer> iOb = new Gen<Integer>(50, n);

        // Нельзя создать массив специфичных для типа обобщенных ссылок.
        // Gen<Integer> gens[] = new Gen<Integer>[10]; // Неверно!

        // Это верно.
        Gen<?> gens[] = new Gen<?>[10]; // OK
    }
}
```

Как показывает эта программа, объявлять ссылку на массив объектов типа `T` допустимо, как это сделано в следующей строке.

```
T vals[]; // OK
```

Тем не менее нельзя создать массив объектов типа `T`, как показано в следующей закомментированной строке.

```
// vals = new T[10]; // нельзя создавать массив объектов T
```

Причина, по которой нельзя создать массив объектов типа `T`, связана с тем, что тип `T` не существует во время выполнения, а потому у компилятора нет способа узнать, массив элементов какого типа в действительности надо создавать.

Однако вы можете передать ссылку на совместимый по типу массив конструктору `Gen()`, когда объект создается, и присвоить эту ссылку `vals`, как это делается в представленной ниже строке программы.

```
vals = nums; // можно присвоить ссылку существующему массиву
```

Это работает, поскольку массив, переданный классу `Gen`, имеет известный тип, который будет тем же типом, что и тип `T` на момент создания объекта.

Обратите внимание на то, что внутри метода `main()` вы не можете объявить массив ссылок на объекты специфического обобщенного типа. То есть строка

```
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Неверно!
```

не компилируется. Массивы специфических обобщенных типов, попросту, недопустимы, так как они могут нарушить безопасность типов.

Вы *можете* создать массив ссылок на обобщенный тип, если используете шаблоны.

```
Gen<?> gens[] = new Gen<?>[10]; // OK
```

Этот подход лучше применения массивов базовых типов, поскольку, по крайней мере, некоторые проверки типа по-прежнему могут быть выполнены компилятором.

Ограничения обобщенных исключений

Обобщенный класс не может расширять класс `Throwable`. Это значит, что вы не сможете создать обобщенные классы исключений.

ЧАСТЬ II

Библиотека Java

ГЛАВА 15 Обработка строк

Краткий обзор обработки строк в Java был приведен в главе 7. В этой главе мы рассмотрим эту тему подробнее. Как и в других языках программирования, в Java *строка* — это последовательность символов. Но в отличие от некоторых других языков, в которых строки реализованы как массивы символов, в Java строки являются объектами класса `String`.

Реализация строк в виде встроенных объектов позволяет Java предоставить полный комплект средств, обеспечивающих удобство управления строками. Например, Java предоставляет методы для сравнения двух строк, поиска подстрок, объединения двух строк и изменения регистра символов в строке. Кроме того, объекты класса `String` могут быть созданы множеством разных способов, что позволяет легко получать строки, когда они требуются.

Что несколько неожиданно, так это тот факт, что, создавая объект класса `String`, вы получаете строку, которая не может быть изменена. Иными словами, как только объект класса `String` создан, вы не можете изменить символы, образующие строку. На первый взгляд это может показаться серьезным ограничением. Однако на самом деле это не так уж важно. Вы можете выполнять со строками любые операции. Особенность в том, что всякий раз, когда вам нужна измененная версия существующей строки, создается новый объект класса `String`, включающий все модификации. Оригинальная строка остается неизменяемой. Этот подход используется потому, что фиксированная, неизменяемая, строка может быть реализована более эффективно, нежели изменяемая. Для тех случаев, когда нужны модифицируемые строки, Java предлагает на выбор два класса `StringBuffer` и `StringBuilder`. Оба содержат строки, которые могут быть изменены после того, как созданы.

Классы `String`, `StringBuffer` и `StringBuilder` определены в пакете `java.lang`. Поэтому они доступны всем программистам автоматически. Все они объявлены с модификатором `final`, следовательно, ни от одного из них нельзя получить подклассы. Это позволяет осуществить некоторую оптимизацию, повышающую производительность общих операций со строками. Все три класса реализуют интерфейс `CharSequence`.

И последнее: когда говорится о том, что строки в объектах класса `String` неизменяемы, это означает, что содержимое экземпляра строки не может быть изменено после его создания. Однако переменная, объявленная как ссылка на класс `String`, в любой момент может быть переназначена так, чтобы указывать на другой объект класса `String`.

Конструкторы строк

Класс `String` поддерживает несколько конструкторов. Чтобы создать пустой объект класса `String`, вызывается стандартный конструктор. Например, следующий оператор создает экземпляр класса `String`, не содержащий в себе символов.

```
String s = new String();
```

Зачастую требуется создать строку, которая содержит начальное значение. Класс `String` предлагает множество конструкторов для этого. Чтобы создать строку, инициализированную массивом символов, используйте следующий конструктор.

```
String(char символы[])
```

Вот примеры.

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

Этот конструктор инициализирует строку `s` символами `"abc"`.

С помощью следующего конструктора в качестве инициализирующей строки вы можете задать диапазон символьного массива.

```
String(char символы[ ], int начИндекс, int количСимволов)
```

Здесь *начИндекс* указывает начало диапазона, а *количСимволов* — количество символов, которые нужно использовать.

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

Это инициализирует строку `s` символами `"cde"`.

Используя конструктор, вы можете создать объект класса `String`, который содержит ту же последовательность символов, что и другой объект класса `String`.

```
String(String стрОбъект)
```

Здесь *стрОбъект* — это объект класса `String`. Рассмотрим следующий пример.

```
// Создать один объект String из другого.
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);
    }
}
```

Вывод этой программы будет выглядеть так.

```
Java
Java
```

Как видите, строки `s1` и `s2` содержат одинаковые значения.

Даже несмотря на то, что тип Java `char` использует 16 бит для представления базового набора символов Unicode, типичный формат строк в Интернете использует массивы 8-битовых байтов, созданных из набора символов ASCII. Поскольку 8-битовые строки ASCII употребляются наиболее часто, класс `String` предлагает конструкторы, которые инициализируют строки массивом типа `byte`. Их форма приведена ниже.

```
String(byte символыAscii[ ])
String(byte символыAscii[ ], int начИндекс, int количСимволов)
```

Здесь параметр *символыAscii* представляет массив байтов. Вторая форма позволяет вам указать диапазон. В каждом из этих конструкторов преобразование байтов в символы выполняется в соответствии с кодировкой по умолчанию для конкретной платформы. Применение этих конструкторов иллюстрируется в следующей программе.

```
// Создание строки из подмножества символьного массива.
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);
    }
}
```

Ниже показан вывод этой программы.

```
ABCDEF
CDE
```

Существуют также расширенные версии конструкторов "байт-строка", в которых вы можете указать кодировку символов, определяющую способ преобразования байтов в символы. Однако, как правило, вам подойдет кодировка для данной платформы по умолчанию.

На заметку! Содержимое массива копируется всякий раз, когда вы создаете объект класса `String` из массива. Если вы модифицируете содержимое массива после создания строки, ваш объект класса `String` останется неизменяемым.

Вы можете создать объект класса `String` из объекта класса `StringBuffer`, используя следующий конструктор.

```
String(StringBuffer объектStrBuf)
```

Вы можете создать строку из объекта класса `StringBuilder`, используя следующий конструктор.

```
String(StringBuilder объектStrBuild)
```

Следующий конструктор поддерживает расширенный набор символов Unicode.

```
String(int кодовыеТочки[], int начИндекс, int количСимволов)
```

Здесь *кодовыеТочки*—массив, содержащий символы Unicode. Результирующая строка создается из диапазона, начинающегося с *начИндекс* и имеющего длину *количСимволов*.

Существуют также конструкторы, позволяющие определять набор символов.

На заметку! Обсуждение элементов кода Unicode и способов работы с ними в Java можно найти в главе 16.

Длина строки

Количество символов, из которых состоит строка, определяет ее длину. Чтобы получить это значение, вызовите метод `length()`.

```
int length()
```

Следующий фрагмент выводит `"3"`, поскольку именно три символа содержит строка `s`.

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Специальные строковые операции

Поскольку строки — общая и важная часть программирования, язык Java обеспечивает специальную поддержку некоторых строковых операций в рамках синтаксиса языка. Эти операции включают автоматическое создание новых экземпляров класса `String` из строковых литералов, конкатенацию множества объектов класса `String` с помощью оператора +, а также преобразование других типов данных в строковое представление. Существуют явные методы для реализации всех этих функций, но язык Java также выполняет их автоматически для удобства программистов и большей ясности.

Строковые литералы

В предыдущих примерах было показано, как явным образом создавать объекты класса `String` из массива символов с помощью оператора `new`. Однако есть более простой способ сделать это с помощью строковых литералов. Для каждого строкового литерала в вашей программе Java автоматически создается объект класса `String`. Таким образом, вы можете использовать строковый литерал для инициализации объекта класса `String`. Например, следующий фрагмент кода создает две эквивалентные строки.

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);

String s2 = "abc"; // используется строковый литерал
```

Поскольку объект класса `String` создается для каждого строкового литерала, вы можете использовать литерал в любом месте, где допускается применение объекта класса `String`. Например, вы можете вызывать методы непосредственно со строками в кавычках, как если бы они были ссылками на объекты, что показано в следующем примере. Здесь вызывается метод `length()` для строки `"abc"`. Как и ожидалось, он возвращает значение 3.

```
System.out.println("abc".length());
```

Конкатенация строк

В общем случае язык Java не позволяет применять операторы к объектам класса `String`. Одно исключение из правил — применение оператора +, который соединяет две строки, порождая в результате объект класса `String`. Это позволяет соединять вместе серии операторов +. Например, в следующем фрагменте кода осуществляется конкатенация трех строк.

```
String age = "9";
String s = "Ему " + age + " лет.";
System.out.println(s);
```

В результате отображается строка `"Ему 9 лет"`.

Одно практическое применение конкатенации строк — когда вы создаете очень длинные строки. Вместо того чтобы включать в код длинные строки одним куском, вы можете разбить их на маленькие фрагменты, используя оператор + для конкатенации.

```
// Использование конкатенации во избежание длинных строк.
class ConCat {
    public static void main(String args[]) {
        String longStr = "Это может быть " +
                         "очень длинная строка, которую следует " +
                         "перенести. Но конкатенация позволяет " +
                         "предотвратить это.";
    System.out.println(longStr);
    }
}
```

Конкатенация строк с другими типами данных

Вы можете соединять строки с данными других типов. Например, рассмотрим слегка измененную версию предыдущего примера.

```
int age = 9;
String s = "Ему " + age + " лет.";
System.out.println(s);
```

В этом случае переменная `age` имеет тип `int`, а не `String`, но результат получается тот же, что и раньше. Так происходит потому, что значение типа `int` автоматически преобразуется в строковое представление в объекте класса `String`. После этого строки конкатенируются, как и прежде. Компилятор преобразует операнды в их строковые эквиваленты, в то время как другие операнды оператора + являются экземплярами класса `String`.

Будьте внимательны, когда смешиваете операнды других типов со строками в выражениях конкатенации. В противном случае можно получить весьма неожиданные результаты. Рассмотрим следующий код.

```
String s = "четыре: " + 2 + 2;
System.out.println(s);
```

Этот фрагмент отображает

```
четыре: 22
вместо
четыре: 4
```

что вы, вероятно, и ожидали. И вот почему. Приоритеты операторов обеспечивают в начале конкатенацию `"четыре"` со строковым эквивалентом числа 2. Результат затем конкатенируется со строковым эквивалентом второго числа 2. Чтобы вначале выполнить целочисленное сложение, следует применить скобки.

```
String s = " четыре: " + (2 + 2);
```

Теперь строка `s` содержит символы `" четыре: 4"`.

Преобразование строк и метод `toString()`

При конкатенации Java преобразует данные в строковое представление вызовом одной из перегруженных версий преобразующих методов `valueOf()`, определенных в классе `String`. Метод `valueOf()` перегружен для всех элементарных типов и для типа `Object`. Для элементарных типов метод `valueOf()` возвращает

строку, которая содержит читабельный для человека эквивалент значения, с которым он был вызван. Для объектов метод `valueOf()` вызывает метод `toString()` этого объекта. Более подробная информация о методе `valueOf()` приведена далее в этой главе. А сейчас давайте изучим метод `toString()` как средство строкового представления объектов классов, которые вы создадите.

Каждый класс реализует метод `toString()`, поскольку этот метод определен в классе `Object`. Однако реализация метода `toString()` по умолчанию редко может быть полезной. Для всех наиболее важных, создаваемых вами классов вы пожелаете переопределить метод `toString()` и обеспечить собственное строковое представление. К счастью, это легко сделать. Общий метод `toString()` имеет следующую форму.

```
String toString()
```

Чтобы реализовать его, просто возвратите объект класса `String`, который содержит читабельную для человека строку, адекватно описывающую объект вашего класса.

Переопределяя метод `toString()` для создаваемых вами классов, вы позволяете им полностью интегрироваться в программное окружение Java. Например, они могут применяться в операторах `print()` и `println()`, а также в строковых выражениях с конкатенацией. Следующая программа демонстрирует это, переопределяя метод `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 " + width + " на " +
        depth + " на " + height + ".";
    }
}
class toStringDemo {
    public static void main(String args[]) {
        Box b = new Box(10, 12, 14);
        String s = "Box b: " + b; // конкатенация объекта Box
        System.out.println(b);    // преобразование Box в строку
        System.out.println(s);
    }
}
```

Вывод этой программы выглядит так.

```
Размеры 10.0 на 14.0 на 12.0
Box b: Размеры 10.0 на 14.0 на 12.0
```

Извлечение символов

Класс `String` предлагает множество способов извлечения символов из объекта класса `String`. Некоторые из них мы рассмотрим. Хотя символы, которые составляют строку, не могут быть индексированы подобно тому, как это делается в символьных массивах, однако многие методы класса `String` используют индек-

сы (или смещения) в строке для выполнения своих операций. Подобно массивам, строки индексируются начиная с нуля.

Метод charAt()

Чтобы выделить из строки единственный символ, вы можете обратиться к нему непосредственно, с помощью метода charAt(), который имеет следующую общую форму.

```
char charAt(int где)
```

Здесь *где* — индекс символа, который нужно получить. Значение *где* должно быть не отрицательным и указывать положение в строке. Метод charAt() возвращает символ, находящийся в указанном положении. Например,

```
char ch;
ch = "abc".charAt(1);
```

присваивает значение "b" переменной ch.

Метод getChars()

Если нужно извлечь более одного символа сразу, вы можете применить метод getChars(), который имеет следующую общую форму.

```
void getChars(int начИсточника, int конИсточника, char цель[], int начЦели)
```

Здесь *начИсточника* указывает индекс начала подстроки, а *конИсточника* — индекс символа, следующего за концом извлекаемой подстроки. Таким образом, извлекается подстрока, содержащая символы от *начИсточника* до *конИсточника*-1. Массив, который принимает выделенные символы, указан в параметре *цель*. Индекс в массиве *цель*, начиная с которого будет записываться подстрока, передается в параметре *начЦели*. Следует позаботиться о том, чтобы массив *цель* был достаточного размера, чтобы в нем поместились все символы указанной подстроки.

В следующей программе демонстрируется использование метода getChars().

```
class getCharsDemo {
    public static void main(String args[]) {
        String s = "Это демонстрация метода getChars.";
        int start = 4;
        int end = 8;
        char buf[] = new char[end - start];
        s.getChars(start, end, buf, 0);
        System.out.println(buf);
    }
}
```

Вывод программы показан ниже.

```
демо
```

Метод getBytes()

Существует альтернатива методу getChars(), которая сохраняет символы в массив байтов. Это метод getBytes(), который использует преобразование символов в байты по умолчанию, предоставляемое платформой. Вот его простейшая форма.

```
byte[] getBytes()
```

Доступны также другие формы этого метода. Метод `getBytes()`, в основном, применим тогда, когда вы экспортируете значения типа `String` в среды, которые не поддерживают 16-битовые символы Unicode. Например, большинство протоколов Интернет и форматов текстовых файлов используют 8-битовый код ASCII для всех текстовых взаимодействий.

Метод `toCharArray()`

Если хотите преобразовать все символы в объекте класса `String` в символьный массив, то простейший способ сделать это — вызвать метод `toCharArray()`. Он возвращает массив символов для всей строки. Его общая форма такова.

```
char toCharArray()
```

Эта функция представлена в качестве дополнения, поскольку тот же результат можно получить, применив метод `getChars()`.

Сравнение строк

Класс `String` включает множество методов, предназначенных для сравнения строк или подстрок в строках. Рассмотрим здесь некоторые из них.

Методы `equals()` и `equalsIgnoreCase()`

Чтобы сравнить две строки на эквивалентность, используйте метод `equals()`. Он имеет следующую общую форму.

```
boolean equals(Object строка)
```

Здесь *строка* — это объект класса `String`, который сравнивается с вызывающим объектом класса `String`. Метод возвращает значение `true`, если строка содержит те же символы и в том же порядке, и значение `false` — в противном случае. Сравнение зависит от регистра.

Чтобы выполнить сравнение, игнорирующее регистр символов, вызовите метод `equalsIgnoreCase()`. Когда этот метод сравнивает две строки, он рассматривает диапазон `A-Z` как то же самое, что и `a-z`. Он имеет следующую общую форму.

```
boolean equalsIgnoreCase(Object строка)
```

Здесь *строка* — это объект класса `String`, который сравнивается с вызывающим объектом класса `String`. Метод также возвращает значение `true`, если строки содержат одинаковые символы в том же порядке, и значение `false` — в противном случае.

Вот пример, демонстрирующий применение методов `equals()` и `equalsIgnoreCase()`.

```
// Демонстрация применения equals() и equalsIgnoreCase().
class equalsDemo {
    public static void main(String args[]) {
        String s1 = "Привет";
        String s2 = "Привет";
        String s3 = "Пока";
        String s4 = "ПРИВЕТ";
        System.out.println(s1 + " эквивалентно " + s2 + " -> " +
                           s1.equals(s2));
        System.out.println(s1 + " эквивалентно " + s3 + " -> " +
                           s1.equals(s3));
```

```
        System.out.println(s1 + " эквивалентно " + s4 + " -> " +
                           s1.equals(s4));
        System.out.println(s1 + " эквивалентно, игнорируя регистр " +
                           s4 + " -> " + s1.equalsIgnoreCase(s4));
    }
}
```

Вывод программы показан ниже.

```
Привет эквивалентно Привет -> true
Привет эквивалентно Good-bye -> false
Привет эквивалентно ПРИВЕТ -> false
Привет эквивалентно, игнорируя регистр ПРИВЕТ -> true
```

Метод `regionMatches()`

Этот метод сравнивает указанную часть строки с другой частью строки. Существует также перегруженная форма, которая игнорирует регистр символов при сравнении. Вот общая форма этих двух методов.

```
boolean regionMatches(int начИндекс, String строка2,
                      int начИндексСтроки2, int количСимволов)

boolean regionMatches(boolean игнорироватьРегистр, int начИндекс,
                      String строка2, int начИндексСтроки2,
                      int количСимволов)
```

В обеих версиях *начИндекс* задает индекс начала диапазона строки вызывающего объекта класса `String`. Строка, подлежащая сравнению, передается в параметре *строка2*. Индекс символа, начиная с которого нужно выполнять сравнение в параметре *строка2*, передается в параметре *начИндексСтроки2*, а длина сравниваемой подстроки — в параметре *количСимволов*. Во второй версии, если параметр *игнорироватьРегистр* содержит значение `true`, регистр символов игнорируется. В противном случае регистр учитывается.

Методы `startsWith()` и `endsWith()`

В классе `String` определены два метода, представляющие собой более или менее специализированные формы метода `regionMatches()`. Метод `startWith()` определяет, начинается ли заданный объект класса `String` с указанной строки. В дополнение метод `endsWith()` определяет, завершается ли объект класса `String` заданным фрагментом. Эти методы имеют следующую общую форму.

```
boolean startsWith(String строка)
boolean endsWith(String строка)
```

Здесь *строка* — фрагмент строки, наличие которого соответственно в начале или конце данного объекта класса `String` нужно проверить. Если он присутствует, возвращается значение `true`, иначе — значение `false`. Например,

```
"Foobar".endsWith("bar")
```

и

```
"Foobar".startsWith("Foo")
```

возвращают значение `true`.

Вторая форма метода `startsWith()`, представленная здесь, позволяет задать начальный пункт.

```
boolean startsWith(String строка, int начИндекс)
```

Здесь *начИндекс* определяет индекс символа в исходной строке, с которого начинается поиск. Например,

```
"Foobar".startsWith("bar", 3)
```

возвращает значение `true`.

Сравнение метода `equals()` и оператора ==

Важно понимать различие между методом `equals()` и оператором ==. Это два разных действия. Как было объяснено, метод `equals()` сравнивает символы внутри объекта класса `String`. Оператор == сравнивает две ссылки на объекты и определяет, ссылаются ли они на один и тот же экземпляр. В следующей программе показано, как два разных объекта класса `String` могут содержать одинаковые символы, но ссылки на эти объекты при сравнении будут не эквивалентными.

```
// equals() против ==
class EqualsNotEqualTo {
    public static void main(String args[]) {
        String s1 = "Привет";
        String s2 = new String(s1);

        System.out.println(s1 + " equals " + s2 + " -> " +
                           s1.equals(s2));

        System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
    }
}
```

Переменная `s1` ссылается на экземпляр класса `String`, созданный присвоением литерала `"Привет"`. Объект, на который ссылается переменная `s2`, создается с использованием переменной `s1` в качестве инициализатора. Таким образом, содержимое обоих объектов класса `String` идентично, но это отличные друг от друга объекты. Это означает, что переменные `s1` и `s2` не ссылаются на один и тот же объект, а потому не равны (при сравнении оператором ==), как доказывает вывод предыдущей программы.

```
Привет equals Привет -> true
Привет == Привет -> false
```

Метод `compareTo()`

Зачастую недостаточно знать, что строки просто идентичны. Для приложений, выполняющих сортировку, нужно знать, какая из строк *меньше*, *равна* или *больше* следующей. Строка меньше другой, если она расположена перед ней в лексикографическом порядке. Строка больше другой, если расположена после нее. Метод по имени `compareTo()` служит этой цели. Он определен интерфейсом `Comparable<T>`, который реализует класс `String`. Метод имеет следующую общую форму.

```
int compareTo(String строка)
```

Здесь *строка* — это объект класса `String`, сравниваемый с вызывающим объектом класса `String`. Возвращаемый результат интерпретируется так, как показано в табл. 15.1.

Таблица 15.1. Возвращаемый результат метода `compareTo()`

Значение	Описание
Меньше нуля	Вызывающая строка меньше строки `str`
Больше нуля	Вызывающая строка больше строки `str`
Нуль	Две строки эквивалентны

Ниже представлен пример программы, которая сортирует массив строк. Программа использует метод `compareTo()` для определения порядка в алгоритме пузырьковой сортировки.

```
// Пузырьковая сортировка объектов String.
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]);
        }
    }
```

Выводом этой программы является список слов.

```
Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to
```

Как можно заметить из вывода этого примера, метод `compareTo()` учитывает регистр букв. Слово `"Now"` идет прежде всех остальных, поскольку начинается с заглавной буквы, а заглавная буква имеет меньшее значение в наборе символов ASCII.

Если вы хотите игнорировать регистр символов при сравнении строк, используйте метод `compareToIgnoreCase()`.

```
int compareToIgnoreCase(String строка)
```

Этот метод возвращает тот же результат, что и метод `compareTo()`, за исключением того, что регистр символов игнорируется. Вы можете попытаться подста-

вить этот метод в предыдущую программу. После этого слово "Now" уже не будет первым в списке.

Поиск строк

Класс String предлагает два метода, которые позволяют выполнять поиск в строке определенного символа или подстроки.

- Метод indexOf() — ищет первое вхождение символа или подстроки.
- Метод lastIndexOf() — ищет последнее вхождение символа или подстроки.

Эти два метода перегружены несколькими разными способами. Во всех случаях эти методы возвращают позицию в строке (индекс), где символ или подстрока была найдена, либо значение -1 — в случае неудачи.

Чтобы найти первое вхождение символа, применяется следующая форма.

```
int indexOf(char символ)
```

Чтобы найти последнее вхождение символа, применяется такая форма.

```
int lastIndexOf(char символ)
```

Здесь *символ* — это символ, который нужно искать.

Чтобы найти первое или последнее вхождение подстроки, применяется следующая форма.

```
int indexOf(String строка)int lastIndexOf(String строка)
```

Здесь *строка* задает искомую подстроку.

Вы можете указать начальную позицию для поиска, воспользовавшись следующими формами.

```
int indexOf(int символ, int начИндекс)
int lastIndexOf(int символ, int начИндекс)
int indexOf(String строка, int начИндекс)
int lastIndexOf(String строка, int начИндекс)
```

Здесь *начИндекс* задает начальную позицию поиска. Для метода indexOf() поиск начинается от *начИндекс* до конца строки, а для метода lastIndexOf() — от *начИндекс* до нуля.

Следующий пример показывает, как использовать различные индексные методы для поиска внутри строки:

```
// Демонстрация использования 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));
    }
}
```

Ниже показан вывод этой программы.

```
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 класса `String` неизменяемы, всякий раз, когда вы хотите их модифицировать, следует либо скопировать их содержимое в объект класса `StringBuffer` или `StringBuilder`, либо воспользоваться одним из следующих методов класса `String`, которые создают новые копии строк с внесенными модификациями. Здесь описаны простейшие из этих методов.

Метод `substring()`

Вы можете извлечь подстроку, используя метод `substring()`. Этот метод имеет две формы. Первая форма такова.

```
String substring(int нач_Индекс)
```

Здесь *нач_Индекс* указывает индекс, с которого начнется подстрока. Эта форма возвращает копию подстроки, которая начинается с позиции *нач_Индекс* и продолжается до завершения вызывающей строки.

Вторая форма метода `substring()` позволяет указать как начальный, так и конечный индексы подстроки.

```
String substring(int нач_Индекс, int кон_Индекс)
```

Здесь *нач_Индекс* указывает индекс, с которого начнется подстрока, а *кон_Индекс* — точку конца подстроки. Возвращаемая строка содержит все символы, от первой позиции и до последней, исключая ее.

В следующей программе метод `substring()` используется для замены в строке всех экземпляров одной подстроки другой.

```
 / Замена 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 { // замена всех совпадающих подстрок
            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);
    }
}
```

Вывод этой программы показан ниже.

```
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.
```

Метод `concat()`

Вы можете соединить две подстроки, используя метод `concat()`, как показано ниже.

```
String concat(String строка)
```

Этот метод создает новый строковый объект, содержащий вызываемую строку, к концу которой добавлено содержимое параметра *строка*. Метод `concat()` выполняет ту же функцию, что и оператор +. Например, следующий код помещает символы `"onetwo"` в строку `s2`.

```
String s1 = "one";
String s2 = s1.concat("two");
```

Код создает тот же результат, что и представленная ниже последовательность.

```
String s1 = "one";
String s2 = s1 + "two";
```

Метод `replace()`

Этот метод имеет две формы. Первая заменяет в исходной строке все вхождения одного символа другим.

```
String replace(char исходный, char замена)
```

Здесь параметр *исходный* задает символ, который должен быть заменен символом *замена*. Возвращается результирующая строка. Например,

```
String s = "Hello".replace('l', 'w');
```

помещает в строку `s` слово `"Hewwo"`. Вторая форма метода `replace()` заменяет одну последовательность символов на другую.

```
String replace(CharSequence исходный, CharSequence замена)
```

Эта форма появилась в J2SE 5.

Метод `trim()`

С помощью этого метода возвращается копия вызывающей строки, из которой удалены все ведущие и завершающие пробелы. Он имеет следующую общую форму.

```
String trim()
```

Вот пример.

```
String s = "    Hello World    ".trim();
```

В результате в строку s будет помещен текст "Hello World".

Метод trim() достаточно удобен в составе других команд. Например, следующая программа приглашает пользователя ввести название штата, а затем отображает название города — столицы штата. Она использует метод trim() для удаления всех предваряющих и завершающих пробелов, которые могут быть непреднамеренно введены пользователем.

```
// Использование trim() для обработки команд.
import java.io.*;
class UseTrim {
    public static void main(String args[])
    throws IOException
    {
        // Создается BufferedReader с использованием System.in
        BufferedReader br = new
        BufferedReader(new InputStreamReader(System.in));
        String str;
        System.out.println("Введите 'стоп' для завершения.");
        System.out.println("Введите штат: ");
        do {
            str = br.readLine();
            str = str.trim(); // удалить пробелы
            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"));
    }
```

Преобразование данных с помощью метода valueOf()

Метод valueOf() преобразует данные из внутреннего представления в читабельную для пользователя форму. Это статический метод, который перегружен в классе String для всех встроенных в Java типов таким образом, что каждый тип может быть правильно преобразован в строку. Метод valueOf() перегружен и для типа Object, поэтому объект типа любого класса, который вы создадите, также может использоваться в качестве аргумента. (Вспомните, что класс Object — суперкласс для всех классов.) Ниже показаны некоторые его формы.

```
static String valueOf(double число)
static String valueOf(long число)
static String valueOf(Object объект)
static String valueOf(char символы[])
```

Как упоминалось ранее, метод valueOf() вызывается тогда, когда требуется строковое представление некоторого другого типа данных — например, при операции конкатенации. Вы можете вызывать этот метод непосредственно с любым типом данных и получать адекватное представление типа String. Все элементарные типы преобразуются в их общее строковое представление. Любой объект, который вы можете передать методу valueOf(), возвратит результат

методу `toString()` объекта. Фактически вы можете просто вызвать метод `toString()` и получить тот же результат.

Для большинства массивов метод `valueOf()` возвращает зашифрованную строку, означающую, что это массив определенного типа. Для массивов типа `char`, однако, создается объект класса `String`, содержащий все символы массива типа `char`. Он имеет следующую форму.

```
static String valueOf(char символы[], int начИндекс, int количСимволов)
```

Здесь *символы* — это массив, который содержит символы, *начИндекс* — начальная позиция в массиве, с которой начинается подстрока, а *количСимволов* указывает длину подстроки.

Изменение регистра символов в строке

Метод `toLowerCase()` преобразует все символы строки из верхнего регистра в нижний. Метод `toUpperCase()` преобразует все символы строки из нижнего регистра в верхний. Небуквенные символы, такие как десятичные цифры, остаются неизменными. Вот простейшая форма этих методов.

```
String toLowerCase()
String toUpperCase()
```

Оба метода возвращают объект класса `String`, содержащий эквивалент вызывающей строки соответственно в нижнем или верхнем регистре. В обоих случаях преобразованием управляет заданный по умолчанию региональный язык.

Ниже показан пример, в котором используются методы `toLowerCase()` и `toUpperCase()`.

```
// Демонстрация toUpperCase() и toLowerCase().
class ChangeCase {
    public static void main(String args[])
    {
        String s = "Это тест.";
        System.out.println("Исходная строка: " + s);
        String upper = s.toUpperCase();
        String lower = s.toLowerCase();
        System.out.println("Верхний регистр: " + upper);
        System.out.println("Нижний регистр: " + lower);
    }
}
```

Вывод приведенной выше программы.

```
Исходная строка: Это тест
Верхний регистр: ЭТО ТЕСТ
Нижний регистр: это тест
```

Еще один момент: предоставляются также перегруженные версии методов `toLowerCase()` и `toUpperCase()`, позволяющие определять объект класса `Locale` для управления преобразованием. В некоторых случаях определение региона может быть очень важным и помочь интернационализировать ваше приложение.

Дополнительные методы класса `String`

В дополнение к методам, перечисленным выше, класс `String` обладает также многими другими методами, включая те, которые перечислены в табл. 15.2.

Таблица 15.2. Дополнительные методы класса `String`

Метод	Описание
`int codePointAt(int i)`	Возвращает точку кода символа Unicode, находящегося в позиции `i`
`int codePointBefore(int i)`	Возвращает точку кода символа Unicode, находящегося в позиции, предшествующей `i`
`int codePointCount(int начало, int конец)`	Возвращает количество точек кода в части вызывающей строки между *начало* и *конец*-1
`boolean contains(CharSequence строка)`	Возвращает значение `true`, если вызывающий объект содержит строку, указанную в *строка*. В противном случае возвращает значение `false`
`boolean contentEquals(CharSequence строка)`	Возвращает значение `true`, если вызывающий объект содержит ту же строку, что и *строка*. В противном случае возвращает значение `false`
`boolean contentEquals(StringBuffer строка)`	Возвращает значение `true`, если вызывающий объект содержит ту же строку, что и *строка*. В противном случае возвращает значение `false`
`static String format(String формстр, Object ... аргументы)`	Возвращает строку, форматированную в соответствии с *формстр*. (Подробности о форматировании описаны в главе 18)
`static String format(Locale регион, String формстр, Object ... аргументы)`	Возвращает строку, форматированную в соответствии с *формстр*. (Подробности о форматировании описаны в главе 18)
`boolean isEmpty()`	Возвращает значение `true`, если вызывающая строка не содержит символов и имеет нулевую длину
`boolean matches(string регВыр)`	Возвращает значение `true`, если вызывающая строка соответствует регулярному выражению, переданному в *регВыр*. В противном случае возвращает значение `false`
`int offsetByCodePoints(int начало, int число)`	Возвращает индекс в вызывающей строке, который находится на *число* точек кода за начальным индексом, указанным в *начало*
`String replaceFirst(String регВыр, String новСтр)`	Возвращает строку, в которой первая подстрока, соответствующая регулярному выражению *регВыр*, заменяется *новСтр*
`String replaceAll(String регВыр, String новСтр)`	Возвращает строку, в которой все подстроки, соответствующие регулярному выражению *регВыр*, заменяются *новСтр*
`String[] split(String регВыр)`	Разбивает вызывающую строку на части и возвращает массив, содержащий результат. Каждая часть ограничена регулярным выражением *регВыр*
`String[] split(String регВыр, int макс)`	Разбивает вызывающую строку на части и возвращает массив, содержащий результат. Каждая часть ограничена регулярным выражением *регВыр*. Количество частей указано в *макс*. Если значение *макс* отрицательное, значит, вызывающая строка разбивается полностью. В противном случае, если *макс* содержит неотрицательное значение, то последний элемент возвращаемого массива содержит остаток вызывающей строки. Если значение *макс* равно нулю, строка также разбивается полностью

Окончание табл. 15.2

Метод	Описание
`CharSequence subSequence(int` *`начИндекс`*`, int` *`конИндекс`*`)`	Возвращает подстроку вызывающей строки, начиная с *начИндекс* и заканчивая *конИндекс*. Этот метод требует интерфейса `CharSequence`, который реализует класс `String`

Обратите внимание на то, что некоторые эти методы работают с регулярными выражениями. Регулярные выражения описаны в главе 28.

Класс `StringBuffer`

Этот класс подобен классу `String`, который представляет большую часть функциональных возможностей строк. Как вы знаете, класс `String` представляет неизменяемые последовательности символов постоянной длины. В отличие от этого, класс `StringBuffer` представляет расширяемые и доступные для изменений последовательности символов. Он позволяет вставлять символы и подстроки в средину либо добавлять их в конец. Объект класса `StringBuffer` автоматически увеличивает размер, чтобы обеспечить место для подобных расширений и зачастую чтобы обеспечить возможность роста, имеет больше предварительно зарезервированных символов, чем фактически нужно в данный момент.

Конструкторы класса `StringBuffer`

В классе `StringBuffer` определены следующие четыре конструктора.

```
StringBuffer()
StringBuffer(int размер)
StringBuffer(String строка)
StringBuffer(CharSequence символы)
```

Стандартный конструктор (без параметров) резервирует место под 16 символов без перераспределения памяти. Вторая версия принимает целочисленный аргумент, который явно устанавливает размер буфера. Третья версия принимает аргумент класса `String`, который устанавливает начальное содержимое объекта класса `StringBuffer` и резервирует 16 символов без повторного резервирования. Класс `StringBuffer` выделяет место под 16 дополнительных символов, когда не указывается конкретный размер буфера, поскольку резервирование памяти — дорогостоящая операция в смысле затрат времени. Кроме того, повторное резервирование может фрагментировать память. Выделяя место под несколько дополнительных символов, класс `StringBuffer` снижает количество необходимых повторных резервирований. Четвертый конструктор создает объект, содержащий последовательность символов из параметра *символы*, а также резервирует место еще для 16 символов.

Методы `length()` и `capacity()`

Текущую длину объекта класса `StringBuffer` можно получить методом `length()`, а текущий объем выделенной памяти — методом `capacity()`. Они имеют следующую общую форму.

```
int length()
int capacity()
```

Ниже показан пример.

```
// Сравнение методов length() и capacity() класса StringBuffer.
class StringBufferDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("buffer = " + sb);
        System.out.println("length = " + sb.length());
        System.out.println("capacity = " + sb.capacity());
    }
}
```

Ниже представлен вывод этой программы, который показывает, как класс `StringBuffer` резервирует запасное пространство для дополнительных манипуляций.

```
buffer = Hello
length = 5
capacity = 21
```

Поскольку строка `sb` инициирована значением `"Hello"` при создании, ее длина равна 5. Объем выделенной памяти (capacity) равен 21, так как 16 дополнительных символов добавлены автоматически.

Метод `ensureCapacity()`

Если вы хотите предварительно выделить место для определенного количества символов после создания объекта класса `StringBuffer`, можете воспользоваться методом `ensureCapacity()`, чтобы установить размер буфера. Это удобно, если знаете наперед, что собираетесь добавлять в объект класса `StringBuffer` большое количество маленьких строк. Метод `ensureCapacity()` имеет следующую общую форму.

```
void ensureCapacity(int минЕмкость)
```

Здесь *минЕмкость* указывает минимальный размер буфера. (Буферы, размер которых превышает *минЕмкость*, также могут быть зарезервированы, по причинам эффективности.)

Метод `setLength()`

Чтобы установить длину строки внутри объекта класса `StringBuffer`, используйте метод `setLength()`. Общая форма этого метода выглядит следующим образом.

```
void setLength(int длина)
```

Здесь *длина* указывает длину строки. Значение должно быть неотрицательным.

Когда вы увеличиваете размер строки, в конец существующей строки добавляются нулевые символы. Если вы вызываете метод `setLength()` со значением, меньшим текущего значения, возвращаемого методом `length()`, то символы, находящиеся за пределами вновь установленной длины, будут утеряны. Пример программы в следующем разделе использует метод `setLength()` для сокращения объекта класса `StringBuffer`.

Методы `charAt()` и `setCharAt()`

Значение отдельного символа может быть извлечено из объекта класса `StringBuffer` методом `charAt()`. Вы можете установить значение символа внутри объекта класса `StringBuffer` с помощью метода `setCharAt()`. Общая форма этих методов показана ниже.

```
char charAt(int где)
void setCharAt(int где, char символ)
```

Для метода `charAt()` параметр *где* указывает индекс символа, который нужно извлечь. Для метода `setCharAt()` параметр *где* указывает индекс символа, который нужно установить, а *символ* — его значение. Значение параметра *где* для обоих методов должно быть неотрицательным и не должно находиться за пределами конца строки.

В следующем примере демонстрируется применение методов `charAt()` и `setCharAt()`.

```
// Демонстрация 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("до charAt(1) = " + sb.charAt(1));

        sb.setCharAt(1, 'i');
        sb.setLength(2);
        System.out.println("буфер после = " + sb);
        System.out.println("после charAt(1) = " + sb.charAt(1));
    }
}
```

Вывод, созданный этой программой, выглядит так.

```
буфер до = Hello
до charAt(1) = e
буфер после = Hi
после charAt(1) = i
```

Метод `getChars()`

Чтобы скопировать подстроку из объекта класса `StringBuffer` в массив, используйте метод `getChars()`. Он имеет следующую форму.

```
void getChars(int начИсточника, int конИсточника, char цель[], int начЦели)
```

Здесь *начИсточника* указывает индекс начала подстроки, а *конИсточника* — индекс символа, следующего за концом требуемой подстроки. Это означает, что подстрока содержит символы от *начИсточника* до *конИсточника*-1. Массив, который принимает символы, передается через параметр *цель*. Индекс внутри параметра *цель*, куда копируется подстрока, передается в параметре *начЦели*. Необходимо позаботиться о том, чтобы массив *цель* был достаточного размера, чтобы вместить количество символов указанной подстроки.

Метод `append()`

Метод `append()` соединяет строковое представление любого другого типа данных с концом вызывающего объекта класса `StringBuffer`. Он имеет несколько перегруженных версий. Вот несколько из них.

```
StringBuffer append(String строка)
StringBuffer append(int число)
StringBuffer append(Object объект)
```

Строковое представление каждого параметра зачастую получают вызовом метода `String.valueOf()`. Результат добавляется к текущему объекту класса `StringBuffer`. Сам буфер возвращается каждой версии метода `append()`. Это

позволяет соединять несколько последовательных вызовов вместе, как показано в следующем примере.

```
 / Демонстрация применения 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);
    }
```

Вывод этого примера показан ниже.

```
a = 42;
```

Метод `insert()`

Этот метод вставляет одну строку в другую. Он перегружен так, чтобы принимать в параметре значения всех элементарных типов плюс объекты классов `String`, `Object` и `CharSequence`. Подобно методу `append()`, он получает строковое представление значения, с которым вызван. Эта строка затем вставляется в вызывающий объект класса `StringBuffer`. Существует несколько форм этого метода.

```
StringBuffer insert(int индекс, String строка)
StringBuffer insert(int индекс, char символ)
StringBuffer insert(int индекс, Object объект)
```

Здесь *индекс* указывает индекс позиции вызывающего объекта класса `StringBuffer`, в которой будет вставлена строка. Следующий пример программы демонстрирует вставку слова `"like"` между `"I"` и `"Java"`:

```
// Демонстрация применения insert().
class insertDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("I Java!");

        sb.insert(2, "like ");
        System.out.println(sb);
    }
}
```

Вывод этой программы выглядит следующим образом.

```
I like Java!
```

Метод `reverse()`

Изменить порядок символов в объекте класса `StringBuffer` на обратный можно с помощью метода `reverse()`.

```
StringBuffer reverse()
```

Этот метод возвращает объект с обратной последовательностью символов по отношению к тому, который его вызвал. В следующей программе демонстрируется использование метода `reverse()`.

```
// Применения reverse() для обращения порядка StringBuffer.
class ReverseDemo {
    public static void main(String args[]) {
        StringBuffer s = new StringBuffer("abcdef");
```

```
        System.out.println(s);
        s.reverse();
        System.out.println(s);
    }
}
```

Вывод этой программы показан ниже.

```
Abcdef
fedcba
```

Методы delete() и deleteCharAt()

Вы можете удалять символы из объекта класса StringBuffer с помощью методов delete() и deleteCharAt().

```
StringBuffer delete(int начИндекс, int конИндекс)
StringBuffer deleteCharAt(int позиция)
```

Метод delete() удаляет последовательность символов из вызывающего объекта. Здесь *начИндекс* задает индекс первого символа, который надо удалить, а *конИндекс* — индекс символа, следующего за последним из удаляемых. Таким образом, удаляемая подстрока начинается с *начИндекс* и заканчивается *конИндекс*-1. Возвращается результирующий объект класса StringBuffer.

Метод deleteCharAt() удаляет символ, находящийся в позиции *позиция*. Возвращает результирующий объект класса StringBuffer.

Вот программа, которая демонстрирует методы delete() и deleteCharAt().

```
// Демонстрация применения delete() и deleteCharAt()
class deleteDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("This is a test.");

        sb.delete(4, 7);
        System.out.println("После delete: " + sb);

        sb.deleteCharAt(0);
        System.out.println("После deleteCharAt: " + sb);
    }
}
```

Программа создает такой вывод.

```
После delete: This a test.
После deleteCharAt: his a test.
```

Метод replace()

Вызвав метод replace(), вы можете заменить один набор символов другим внутри объекта класса StringBuffer. Сигнатура этого метода показана ниже.

```
StringBuffer replace(int начИндекс, int конИндекс, String строка)
```

Подстрока, которую нужно заменить, задается индексами *начИндекс* и *конИндекс*. Таким образом, заменяется подстрока от символа *начИндекс* до *конИндекс*-1. Строка замены передается в параметре *строка*. Возвращается результирующий объект класса StringBuffer.

В следующей программе демонстрируется использование метода replace().

```
// Демонстрация применения replace()
class replaceDemo {
```

```
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("This is a test.");
        sb.replace(5, 7, "was");
        System.out.println("После замены: " + sb);
    }
}
```

Ниже показан вывод программы.

```
После замены: This was a test.
```

Метод substring()

Вы можете получить часть содержимого объекта класса StringBuffer вызовом метода substring(). Этот метод имеет две следующие формы.

```
String substring(int начИндекс)
String substring(int начИндекс, int конИндекс)
```

Первая форма возвращает подстроку, которая начинается от *начИндекс* и продолжается до конца вызывающего объекта класса StringBuffer. Вторая форма возвращает подстроку от позиции *начИндекс* до *конИндекс*-1. Эти методы работают точно так же, как их описанные выше аналоги в классе String.

Дополнительные методы класса StringBuffer

В дополнение к описанным методам класса StringBuffer, он содержит ряд других, включая перечисленные в табл. 15.3.

Таблица 15.3. Дополнительные методы класса StringBuffer

Метод	Описание
StringBuffer appendCodePoint(int *символ*)	Добавляет точку кода Unicode в конец вызывающего объекта. Возвращается ссылка на объект
int codePointAt(int *i*)	Возвращает точку кода Unicode в позиции, указанной в параметре *i*
int codePointBefore(int *i*)	Возвращает точку кода Unicode в позиции, предшествующей *i*
int codePointCount(int *начало*, int *конец*)	Возвращает количество точек кода в части вызывающей строки, заключенной между *начало* и *конец*-1
int indexOf(String *строка*)	Выполняет поиск в вызывающем объекте класса StringBuffer первого вхождения *строка*. Возвращает индекс позиции совпадения или значение -1 — в случае неудачи
int indexOf(String *строка*, int *начИндекс*)	Выполняет поиск в вызывающем объекте класса StringBuffer первого вхождения *строка* начиная с *начИндекс*. Возвращает индекс позиции совпадения или значение -1 — в случае неудачи
int lastIndexOf(String *строка*)	Выполняет поиск в вызывающем объекте класса StringBuffer последнего вхождения *строка*. Возвращает индекс позиции совпадения или значение -1 — в случае неудачи

Окончание табл. 15.3

Метод	Описание
`int lastIndexOf(String` *`строка`*`, int` *`начИндекс`*`)`	Выполняет поиск в вызывающем объекте класса `StringBuffer` последнего вхождения *строка* начиная с *начИндекс*. Возвращает индекс позиции совпадения или значение `-1` — в случае неудачи
`int offsetByCodePoints(int` *`начало`*`, int` *`число`*`)`	Возвращает индекс символа в вызывающей строке, который находится на *число* точек кода позади начального индекса, указанного в *начало*
`CharSequence subSequence(int` *`начИндекс`*`, int` *`конИндекс`*`)`	Возвращает подстроку вызывающей строки, начиная с *начИндекс* и заканчивая *конИндекс*. Этот метод требует интерфейса `CharSequence`, который реализует класс `StringBuffer`
`void trimToSize()`	Требует, чтобы размер символьного буфера вызывающего объекта был уменьшен для лучшего соответствия текущему содержимому

В следующей программе демонстрируется применение методов `indexOf()` и `lastIndexOf()`.

```
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);
    }
}
```

Вывод будет таким.

```
Индекс первого вхождения: 0
Индекс последнего вхождения: 8
```

Класс `StringBuilder`

Появившейся в JDK 5 класс `StringBuilder` — это дополнение к существующим возможностям обработки строк Java. Класс `StringBuilder` идентичен классу `StringBuffer`, за исключением одного важного отличия: он не синхронизирован, а значит, не является безопасным в отношении потоков. Выгода от применения класса `StringBuilder` связана с более высокой производительностью. Однако в случаях, когда к изменяемой строке обращаются несколько потоков и не используется никакой внешней синхронизации, следует использовать класс `StringBuffer`, а не класс `StringBuilder`.

ГЛАВА 16

Пакет java.lang

Настоящая глава посвящена классам и интерфейсам, определенным в пакете java.lang. Как вы знаете, пакет java.lang автоматически импортируется во все программы. Он содержит классы и интерфейсы, которые являются фундаментальными для всех программ на языке Java. Это наиболее широко используемый пакет Java. Пакет java.lang включает следующие классы.

Boolean	Enum	Process	String
Byte	Float	ProcessBuilder	StringBuffer
Character	Inheritable-ThreadLocal	ProcessBuilder.Redirect	StringBuilder
Character.Subset	Integer	Runtime	System
Character.UnicodeBlock	Long	RuntimePermission	Thread
Class	Math	SecurityManager	ThreadGroup
ClassLoader	Number	Short	ThreadLocal
ClassValue	Object	StackTraceElement	Throwable
Compiler	Package	StrictMath	Void
Double			

Также определены два вложенных класса Character: Character.SubSet и Character.UnicodeBlock.

В пакете java.lang определены следующие интерфейсы.

Appendable	Cloneable	Readable
AutoCloseable	Comparable	Runnable
CharSequence	Iterable	Thread.UncaughtExceptionHandler

Некоторые классы, включенные в пакет java.lang, содержат устаревшие (deprecated) методы, большинство из которых относится еще к Java 1.0. Эти устаревшие методы все еще предоставляются языком Java для поддержки постепенно отмирающего унаследованного кода и не рекомендуются для применения в новом коде. Большинство устаревших элементов существовало до версии JDK 7, и эти методы здесь не обсуждаются.

Оболочки элементарных типов

Как упоминалось в части I настоящей книги, язык Java использует элементарные типы, такие как `int` и `char`, из соображений производительности. Эти типы данных не являются частью объектной иерархии. Они передаются по значению в методы и не могут быть переданы по ссылке. Также нет способа для двух методов ссылаться на *один и тот же экземпляр* типа `int`. Рано или поздно у вас возникнет необходимость в объектном представлении одного из элементарных типов. Например, существуют классы коллекций, описанные в главе 17, которые имеют дело только с объектами. Чтобы сохранить элементарный тип в одном из этих классов, необходимо поместить элементарный тип в оболочку класса. Чтобы удовлетворить эту потребность, язык Java предлагает классы, которые соответствуют каждому элементарному типу. По сути, эти классы инкапсулируют элементарные типы в классы (или, что то же самое, помещают элементарные типы в *оболочки* классов). Таким образом, их обычно называют оболочками типов. Оболочки типов были впервые представлены в главе 12. Здесь мы рассмотрим их более подробно.

Класс `Number`

Абстрактный класс `Number` — это суперкласс, который реализуют классы, являющиеся оболочками для числовых типов `byte`, `short`, `int`, `long`, `float` и `double`. Класс `Number` имеет абстрактные методы, которые возвращают значение объекта соответствующего числового типа. Например, метод `doubleValue()` возвращает значение как тип `double`, а метод `floatValue()` — как тип `float` и т.д. Эти методы перечислены ниже.

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

Значения, возвращаемые этими методами, могут быть округлены. Усечение также возможно.

Класс `Number` имеет конкретные подклассы, которые содержат явные значения каждого числового типа: `Double`, `Float`, `Byte`, `Short`, `Integer` и `Long`.

Классы `Double` и `Float`

Эти классы являются оболочками для значений с плавающей точкой типов `double` и `float` соответственно. Конструкторы класса `Float` показаны ниже.

```
Float(double число)
Float(float число)
Float(String строка) throws NumberFormatException
```

Как вы можете видеть, объекты класса `Float` должны быть созданы со значениями типа `float` или `double`. Они могут также быть созданы из строкового представления числа с плавающей точкой.

Вот как выглядят конструкторы класса `Double`.

```
Double(double число)
Double(String строка) throws NumberFormatException
```

Объекты класса `Double` могут быть созданы из значения типа `double` или строки, содержащей значение с плавающей точкой.

Методы, определенные в классе Float, описаны в табл. 16.1. Методы, определенные в классе Double, перечислены в табл. 16.2. Классы Float и Double определяют следующие константы.

MAX_EXPONENT	Максимальная экспонента
MAX_VALUE	Максимальное положительное значение
MIN_EXPONENT	Минимальная экспонента
MIN_NORMAL	Минимальное положительное нормальное значение
MIN_VALUE	Минимальное положительное значение
NaN	Не число
POSITIVE_INFINITY	Положительная бесконечность
NEGATIVE_INFINITY	Отрицательная бесконечность
SIZE	Размер помещенного в оболочку значения в битах
TYPE	Объект класса Class для типов float и double

Таблица 16.1. Методы класса Float

Метод	Описание
byte byteValue()	Возвращает значение вызывающего объекта как тип byte
static int compare(float *число1*, float *число2*)	Сравнивает значения *число1* и *число2*. Возвращает 0, если значения эквивалентны. Возвращает отрицательное значение, если *число1* меньше *число2*, и положительное — если *число1* больше *число2*
int compareTo(Float *f*)	Сравнивает числовое значение вызывающего объекта со значением *f*. Возвращает 0, если значения эквивалентны. Возвращает отрицательное значение, если вызывающий объект имеет меньшее значение. Возвращает положительное значение, если вызывающий объект имеет большее значение
double doubleValue()	Возвращает значение вызывающего объекта как тип double
boolean equals(Object *объектFloat*)	Возвращает значение true, если вызывающий объект класса Float эквивалентен *объектFloat*. В противном случае возвращает значение false
static int floatToIntBits(float *число*)	Возвращает совместимый с IEEE битовый шаблон одинарной точности, который соответствует *число*
static int floatToRawIntBits(float *число*)	Возвращает совместимый с IEEE битовый шаблон одинарной точности, который соответствует *число*. Значения NaN не допускаются
float floatValue()	Возвращает значение вызывающего объекта как float
int hashCode()	Возвращает хеш-код вызывающего объекта
static float intBitsToFloat(int *число*)	Возвращает эквивалент float совместимого с IEEE битового шаблона одинарной точности, который соответствует *число*

Окончание табл. 16.1

Метод	Описание
`int intValue()`	Возвращает значение вызывающего объекта как тип `int`
`boolean isInfinite()`	Возвращает значение `true`, если вызывающий объект содержит бесконечное значение. В противном случае возвращает значение `false`
`static boolean isInfinite(float` *число*`)`	Возвращает значение `true`, если *число* определяет бесконечное значение. В противном случае возвращает значение `false`
`boolean isNaN()`	Возвращает значение `true`, если вызывающий объект содержит значение, которое не является числом. В противном случае возвращает значение `false`
`static boolean isNaN(float` *число*`)`	Возвращает значение `true`, если *число* определяет значение, не являющееся числом. В противном случае возвращает значение `false`
`long longValue()`	Возвращает значение вызывающего объекта как тип `long`
`static float parseFloat(String` *строка*`) throws NumberFormatException`	Возвращает эквивалент типа `float` числа с основанием 10, содержащегося в строке *строка*
`short shortValue()`	Возвращает значение вызывающего объекта как тип `short`
`static String toHexString(float` *число*`)`	Возвращает строку, содержащую значение *число* в шестнадцатеричном формате
`String toString()`	Возвращает строковый эквивалент вызывающего объекта
`static String toString(float` *число*`)`	Возвращает строковый эквивалент значения, представленного параметром *число*
`static Float valueOf(float` *число*`)`	Возвращает объект `Float`, содержащий значение, переданное параметром *число*
`static Float valueOf(String` *строка*`) throws NumberFormatException`	Возвращает объект `Float`, содержащий значение, представленное в строке *строка*

Таблица 16.2. Методы класса `Double`

Метод	Описание
`byte byteValue()`	Возвращает значение вызывающего объекта как `byte`
`static int compare(double` *число1*`, double` *число2*`)`	Сравнивает значения *число1* и *число2*. Возвращает 0, если значения эквивалентны. Возвращает отрицательное значение, если *число1* меньше *число2*. Возвращает положительное значение, если *число1* больше *число2*

Продолжение табл. 16.2

Метод	Описание
`int compareTo(Double d)`	Сравнивает числовое значение вызывающего объекта с *d*. Возвращает 0, если значения эквивалентны. Возвращает отрицательное значение, если вызывающий объект имеет меньшее значение. Возвращает положительное значение, если вызывающий объект имеет большее значение
`static long doubleToLongBits(double число)`	Возвращает совместимый с IEEE битовый шаблон двойной точности, который соответствует параметру *число*
`static long doubleToRawLongBits(double число)`	Возвращает совместимый с IEEE битовый шаблон двойной точности, который соответствует параметру *число*. Значения NaN не допускаются
`double doubleValue()`	Возвращает значение вызывающего объекта как тип `double`
`boolean equals(Object объектDouble)`	Возвращает значение `true`, если вызывающий объект класса `Double` эквивалентен *объектDouble*
`float floatValue()`	Возвращает значение вызывающего объекта как тип `float`
`int hashcode()`	Возвращает хеш-код вызывающего объекта
`int intValue()`	Возвращает значение вызывающего объекта как тип `int`
`boolean isInfinite()`	Возвращает значение `true`, если вызывающий объект содержит бесконечное значение. В противном случае возвращает значение `false`
`static boolean isInfinite(double число)`	Возвращает значение `true`, если *число* определяет бесконечное значение. В противном случае возвращает значение `false`
`boolean isNaN()`	Возвращает значение `true`, если вызывающий объект содержит нечисловое значение. В противном случае возвращает значение `false`
`static boolean isNaN(double число)`	Возвращает значение `true`, если *число* определяет нечисловое значение. В противном случае возвращает значение `false`
`static double longBitsToDouble(long число)`	Возвращает эквивалент типа `double` совместимого с IEEE битового шаблона, заданного параметром *число*
`long longValue()`	Возвращает значение вызывающего объекта как тип `long`
`static double parseDouble(String строка) throws NumberFormatException`	Возвращает эквивалент типа `double` числа с основанием 10, содержащегося в строке *строка*

Окончание табл. 16.2

Метод	Описание
`short shortValue()`	Возвращает значение вызывающего объекта как тип `short`
`static String toHexString(double` *число*`)`	Возвращает строку, содержащую значение *число* в шестнадцатеричном формате
`String toString()`	Возвращает строковый эквивалент вызывающего объекта
`static String toString(double` *число*`)`	Возвращает строковый эквивалент значения, заданного параметром *число*
`static Double valueOf(double` *число*`)`	Возвращает объект класса `Double`, содержащий значение, переданное в параметре *число*
`static Double valueOf(String` *строка*`) throws NumberFormatException`	Возвращает объект класса `Double`, содержащий значение, переданное в строке *строка*

В следующем примере создаются два объекта класса `Double` — с использованием значения типа `double` и с использованием строки, которая может быть интерпретирована как тип `double`.

```
class DoubleDemo {
    public static void main(String args[]) {
        Double d1 = new Double(3.14159);
        Double d2 = new Double("314159E-5");

        System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
    }
}
```

Как можно заметить из следующего вывода, оба конструктора создают идентичные экземпляры класса `Double`, что доказывает метод `equals()`, возвращающий значение `true`.

```
3.14159 = 3.14159 -> true
```

Методы `isInfinite()` и `isNaN()`

Классы `Float` и `Double` предлагают методы `isInfinite()` и `isNaN()`, которые помогают манипулировать двумя специальными значениями типов `double` и `float`. Эти методы осуществляют проверку на предмет равенства двум уникальным значениям, определенным спецификациями стандарта IEEE для чисел с плавающей точкой, — бесконечности и `NaN` (не число). Метод `isInfinite()` возвращает значение `true`, если проверяемое число бесконечно велико или бесконечно мало по величине. Метод `isNaN()` возвращает значение `true`, если проверяемое значение не является числовым.

В следующем примере создаются два объекта класса `Double`: один содержит бесконечность, а второй — нечисловое значение.

```
// Демонстрация применения isInfinite() и isNaN()
class InfNaN {
    public static void main(String args[]) {
        Double d1 = new Double(1/0.);
```

```
        Double d2 = new Double(0/0.);

        System.out.println(d1 + ": " + d1.isInfinite() + ", " +
                           d1.isNaN());
        System.out.println(d2 + ": " + d2.isInfinite() + ", " +
                           d2.isNaN());
    }
}
```

Программа создает следующий вывод.

```
Infinity: true, false
NaN: false, true
```

Классы Byte, Short, Integer и Long

Классы Byte, Short, Integer и Long — это оболочки для целочисленных типов byte, short, int и long соответственно. Ниже показаны их конструкторы.

```
Byte(byte число)
Byte(String строка) throws NumberFormatException

Short(short число)
Short(String строка) throws NumberFormatException

Integer(int число)
Integer(String строка) throws NumberFormatException

Long(long число)
Long(String строка) throws NumberFormatException
```

Как видите, эти объекты могут быть созданы из числовых значений или строк, которые содержат допустимые представления числовых значений.

Методы, определенные этими двумя классами, показаны в табл. 16.3–16.6. Как вы можете видеть, они определяют методы для разбора целых чисел из строк и преобразования строк обратно в целые числа. Вариации этих методов позволяют вам указать *основание* (radix) — основание числа для преобразования. Чаще всего применяются основание 2 — для двоичных чисел, 8 — для восьмеричных, 10 — для десятичных и 16 — для шестнадцатеричных чисел.

В этих классах определены следующие константы.

MIN_VALUE	Минимальное значение
MAX_VALUE	Максимальное значение
SIZE	Ширина помещенного в оболочку значения в битах
TYPE	Объект Clas для типов byte, short, int или long

Таблица 16.3. Методы класса Byte

Метод	Описание
byte byteValue()	Возвращает значение вызывающего объекта как тип byte
static int compare(byte *число1*, byte *число2*)	Сравнивает значения *число1* и *число2*. Возвращает 0, если значения равны. Возвращает отрицательное значение, если значение *число1* меньше, чем *число2*. Возвращает положительное значение, если значение *число1* больше, чем *число2*. (Добавлено в JDK 7)

Окончание табл. 16.3

Метод	Описание
`int compareTo(Byte b)`	Сравнивает числовое значение вызывающего объекта с *b*. Возвращает 0, если значения равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение. Возвращает положительное число, если вызывающий объект имеет большее значение
`static Byte decode(String строка) throws NumberFormatException`	Возвращает объект класса `Byte`, который содержит значение, указанное в строке *строка*
`double doubleValue()`	Возвращает значение вызывающего объекта как тип `double`
`boolean equals(Object объектByte)`	Возвращает значение `true`, если вызывающий объект класса `Byte` эквивалентен *объектByte*. В противном случае возвращает значение `false`
`float floatValue()`	Возвращает значение вызывающего объекта как значение типа `float`
`int hashCode()`	Возвращает хеш-код вызывающего объекта
`int intValue()`	Возвращает значение вызывающего объекта как тип `int`
`long longValue()`	Возвращает значение вызывающего объекта как тип `long`
`static byte parseByte(String строка) throws NumberFormatException`	Возвращает эквивалент типа `byte` числа с основанием 10, переданного в строке *строка*
`static byte parseByte(String строка, int основание) throws NumberFormatException`	Возвращает эквивалент типа `byte` числа с основанием *основание*, переданного в строке *строка*
`short shortValue()`	Возвращает значение вызывающего объекта как `short`
`String toString()`	Возвращает строку, которая содержит десятичный эквивалент вызывающего объекта
`static String toString(byte число)`	Возвращает строку, которая содержит десятичный эквивалент *число*
`static Byte valueOf(byte число)`	Возвращает объект класса `Byte`, содержащий значение, переданное параметром *число*
`static Byte valueOf(String строка) throws NumberFormatException`	Возвращает объект класса `Byte`, содержащий значение, заданное в строке *строка*
`static Byte valueOf(String строка, int основание) throws NumberFormatException`	Возвращает объект класса `Byte`, содержащий значение, заданное в строке *строка* с использованием основания *основание*

Таблица 16.4. Методы класса `Short`

Метод	Описание
`byte byteValue()`	Возвращает значение вызывающего объекта как тип `byte`

Окончание табл. 16.4

Метод	Описание
`static int compare(short число1, short число2)`	Сравнивает значения *число1* и *число2*. Возвращает 0, если значения равны. Возвращает отрицательное значение, если значение *число1* меньше, чем *число2*. Возвращает положительное значение, если значение *число1* больше, чем *число2*. (Добавлено в JDK 7)
`int compareTo(Short s)`	Сравнивает числовое значение вызывающего объекта с *s*. Возвращает 0, если значения равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение. Возвращает положительное число, если вызывающий объект имеет большее значение
`static Short decode(String строка) throws NumberFormatException`	Возвращает объект класса `Short`, который содержит значение, указанное в строке *строка*
`double doubleValue()`	Возвращает значение вызывающего объекта как тип `double`
`boolean equals(Object объектShort)`	Возвращает значение `true`, если вызывающий объект класса `Short` эквивалентен *объектShort*. В противном случае возвращает значение `false`
`float floatValue()`	Возвращает значение вызывающего объекта как тип `float`
`int hashCode()`	Возвращает хеш-код вызывающего объекта
`int intValue()`	Возвращает значение вызывающего объекта как тип `int`
`long longValue()`	Возвращает значение вызывающего объекта как тип `long`
`static short parseShort(String строка) throws NumberFormatException`	Возвращает эквивалент типа `short` числа с основанием 10, переданного в строке *строка*
`static short parseShort(String строка, int основание) throws NumberFormatException`	Возвращает эквивалент типа `short` числа с основанием *основание*, переданного в строке *строка*
`static short reverseBytes(short число)`	Меняет местами старший и младший байты *число* и возвращает результат
`short shortValue()`	Возвращает значение вызывающего объекта как тип `short`
`String toString()`	Возвращает строку, которая содержит десятичный эквивалент вызывающего объекта
`static String toString(short число)`	Возвращает строку, которая содержит десятичный эквивалент *число*
`static Short valueOf(short число)`	Возвращает объект класса `Short`, содержащий значение, переданное в *число*
`static Short valueOf(String строка) throws NumberFormatException`	Возвращает объект класса `Short`, содержащий значение, заданное в строке *строка*
`static Short valueOf(String строка, int основание) throws NumberFormatException`	Возвращает объект класса `Short`, содержащий значение, заданное в строке *строка* с использованием основания *основание*

Таблица 16.5. Методы класса `Integer`

Метод	Описание
`static int bitCount(int` *число*`)`	Возвращает количество битов в *число*
`byte byteValue()`	Возвращает значение вызывающего объекта как тип `byte`
`static int compare(int` *число1*`, int` *число2*`)`	Сравнивает значения *число1* и *число2*. Возвращает 0, если значения равны. Возвращает отрицательное значение, если значение *число1* меньше, чем *число2*. Возвращает положительное значение, если значение *число1* больше, чем *число2*. (Добавлено в JDK 7)
`int compareTo(Integer` *i*`)`	Сравнивает числовое значение вызывающего объекта с *i*. Возвращает 0, если значения равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение. Возвращает положительное число, если вызывающий объект имеет большее значение
`static Short decode(String` *строка*`) throws NumberFormatException`	Возвращает объект класса `Short`, который содержит значение, указанное в строке *строка*
`double doubleValue()`	Возвращает значение вызывающего объекта как тип `double`
`boolean equals(Object` *объектInteger*`)`	Возвращает значение `true`, если вызывающий объект эквивалентен *объектInteger*. В противном случае возвращает значение `false`
`float floatValue()`	Возвращает значение вызывающего объекта как тип `float`
`static Integer getInteger(String propertyName)`	Возвращает значение, ассоциированное со свойством окружения, указанным в *имяСвойства*. В случае неудачи возвращается значение `null`
`static Integer getInteger(String` *имяСвойства*`, int` *поумолчанию*`)`	Возвращает значение, ассоциированное со свойством окружения, указанным в *имяСвойства*. В случае неудачи возвращается значение *поумолчанию*
`static Integer getInteger(String` *имяСвойства*`, Integer` *поумолчанию*`)`	Возвращает значение, ассоциированное со свойством окружения, указанным в *имяСвойства*. В случае неудачи возвращается значение *поумолчанию*
`int hashCode()`	Возвращает хеш-код вызывающего объекта
`static int highestOneBit(int` *число*`)`	Определяет позицию самого старшего бита в *число*. Возвращает значение, в котором установлен только этот бит. Если ни одного бита не установлено, возвращается нуль
`int intValue()`	Возвращает значение вызывающего объекта как тип `int`
`long longValue()`	Возвращает значение вызывающего объекта как тип `long`
`static int lowestOneBit(int` *число*`)`	Определяет позицию самого младшего бита в *число*. Возвращает значение, в котором установлен только этот бит. Если установленных бит нет, возвращается нуль

Продолжение табл. 16.5

Метод	Описание
`static int numberOfLeadingZeros(int число)`	Возвращает количество старших битов, установленных в нуль, которые предшествуют первому установленному старшему биту в *число*. Если *число* равно 0, возвращается 32
`static int numberOfTrailingZeros(int число)`	Возвращает количество младших битов, установленных в нуль, которые предшествуют первому установленному младшему биту в *число*. Если *число* равно 0, возвращается 32
`static int parseInt(String строка) throws NumberFormatException`	Возвращает целочисленный эквивалент числа с основанием 10, переданного в строке *строка*
`static int parseInt(String строка, int основание) throws NumberFormatException`	Возвращает целочисленный эквивалент числа с основанием *основание*, переданного в строке *строка*
`static int reverse(int число)`	Изменяет порядок битов в *число* на противоположный и возвращает результат
`static int reverseBytes(int число)`	Изменяет порядок байтов в *число* на противоположный и возвращает результат
`static int rotateLeft(int число, int n)`	Возвращает результат смещения *число* на *n* позиций влево
`static int rotateRight(int число, int n)`	Возвращает результат смещения *число* на *n* позиций вправо
`static int signum(int число)`	Возвращает –1, если *число* отрицательное, 0 – если нуль и 1 – если положительное
`short shortValue()`	Возвращает значение вызывающего объекта как тип `short`
`static String toBinaryString(int число)`	Возвращает строку, содержащую двоичный эквивалент числа *число*
`static String toHexString(int число)`	Возвращает строку, содержащую шестнадцатеричный эквивалент числа *число*
`static String toOctalString(int число)`	Возвращает строку, содержащую восьмеричный эквивалент числа *число*
`String toString()`	Возвращает строку, которая содержит десятичный эквивалент вызывающего объекта
`static String toString(int число)`	Возвращает строку, которая содержит десятичный эквивалент числа *число*
`static String toString(int число, int основание)`	Возвращает строку, которая содержит десятичный эквивалент значения, заданного в строке *строка* с использованием основания *основание*
`static Integer valueOf(int число)`	Возвращает объект класса `Integer`, содержащий значение, переданное в *число*

Окончание табл. 16.5

Метод	Описание
`static Integer valueOf(String строка) throws NumberFormatException`	Возвращает объект класса `Integer`, содержащий значение, заданное в строке *строка*
`static Integer valueOf(String строка, int основание) throws NumberFormatException`	Возвращает объект класса `Integer`, содержащий значение, заданное в строке *строка* с использованием основания *основание*

Таблица 16.6. Методы класса `Long`

Метод	Описание
`static int bitCount(long число)`	Возвращает количество битов в числе *число*
`byte byteValue()`	Возвращает значение вызывающего объекта как тип `byte`
`static int compare(long число1, long число2)`	Сравнивает значения *число1* и *число2*. Возвращает 0, если значения равны. Возвращает отрицательное значение, если значение *число1* меньше, чем *число2*. Возвращает положительное значение, если значение *число1* больше, чем *число2*. (Добавлено в JDK 7)
`int compareTo(Long l)`	Сравнивает числовое значение вызывающего объекта с *l*. Возвращает 0, если значения равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение. Возвращает положительное число, если вызывающий объект имеет большее значение
`static Long decode(String строка) throws NumberFormatException`	Возвращает объект класса `Long`, который содержит значение, указанное в строке *строка*
`double doubleValue()`	Возвращает значение вызывающего объекта как тип `double`
`boolean equals(Object объектLong)`	Возвращает значение `true`, если вызывающий объект класса `Long` эквивалентен *объектLong*. В противном случае возвращает значение `false`
`float floatValue()`	Возвращает значение вызывающего объекта как тип `float`
`static Long getLong(String имяСвойства)`	Возвращает значение, ассоциированное со свойством окружения, указанным в *имяСвойства*. В случае неудачи возвращается значение `null`
`static Long getLong(String имяСвойства, long поумолчанию)`	Возвращает значение, ассоциированное со свойством окружения, указанным в *имяСвойства*. В случае неудачи возвращается значение *поумолчанию*
`static Long getLong(String имяСвойства, Long поумолчанию)`	Возвращает значение, ассоциированное со свойством окружения, указанным в *имяСвойства*. В случае неудачи возвращается значение *поумолчанию*
`int hashCode()`	Возвращает хеш-код вызывающего объекта

Продолжение табл. 16.6

Метод	Описание
`static int highestOneBit(long число)`	Определяет позицию самого старшего бита в числе *число*. Возвращает значение, в котором установлен только этот бит. Если ни одного бита не установлено, возвращается нуль
`int intValue()`	Возвращает значение вызывающего объекта как тип `int`
`long longValue()`	Возвращает значение вызывающего объекта как тип `long`
`static int lowestOneBit(long число)`	Определяет позицию самого младшего бита в числе *число*. Возвращает значение, в котором установлен только этот бит. Если ни одного бита не установлено, возвращается нуль
`static int numberOfLeadingZeros(long число)`	Возвращает количество старших битов, установленных в нуль, которые предшествуют первому установленному старшему биту в *число*. Если *число* равно 0, возвращается 64
`static int numberOfTrailingZeros(long число)`	Возвращает количество младших битов, установленных в нуль, которые предшествуют первому установленному младшему биту в *число*. Если *число* равно 0, возвращается 64
`static long parseLong(String строка) throws NumberFormatException`	Возвращает эквивалент типа `long` числа с основанием 10, переданного в строке *строка*
`static long parseInt(String строка, int основание) throws NumberFormatException`	Возвращает эквивалент `long` числа с основанием *основание*, переданного в строке *строка*
`static long reverse(long число)`	Изменяет порядок битов в числе *число* на противоположный и возвращает результат
`static long reverseBytes(long число)`	Изменяет порядок байтов в числе *число* на противоположный и возвращает результат
`static long rotateLeft(long число, int n)`	Возвращает результат смещения числа *число* на *n* позиций влево
`static long rotateRight(long число, int n)`	Возвращает результат смещения числа *число* на *n* позиций вправо
`static int signum(int число)`	Возвращает -1, если *число* отрицательное, 0 — если нуль и 1 — если положительное
`short shortValue()`	Возвращает значение вызывающего объекта как тип `short`
`static String toBinaryString(long число)`	Возвращает строку, содержащую бинарный эквивалент числа *число*
`static String toHexString(long число)`	Возвращает строку, содержащую шестнадцатеричный эквивалент числа *число*
`static String toOctalString(long число)`	Возвращает строку, содержащую восьмеричный эквивалент числа *число*

Окончание табл. 16.6

Метод	Описание
`String toString()`	Возвращает строку, которая содержит десятичный эквивалент вызывающего объекта
`static String toString(long` *число*`)`	Возвращает строку, которая содержит десятичный эквивалент числа *число*
`static String toString(long` *число*`, int` *основание*`)`	Возвращает строку, которая содержит десятичный эквивалент значения, заданного в строке *строка* с использованием основания *основание*
`static Long valueOf(long` *число*`)`	Возвращает объект класса `Long`, содержащий значение, переданное в параметре *число*
`static Long valueOf(String` *строка*`) throws NumberFormatException`	Возвращает объект класса `Long`, содержащий значение, заданное в строке *строка*
`static Long valueOf(String` *строка*`, int` *основание*`) throws NumberFormatException`	Возвращает объект класса `Long`, содержащий значение, заданное в строке *строка* с использованием основания *основание*

Преобразование чисел в строки и обратно

Одной из наиболее часто выполняемых рутинных операций в программировании является преобразование строкового представления чисел во внутренний двоичный формат. К счастью, в языке Java имеется простой способ осуществления этого. Классы `Byte`, `Short`, `Integer` и `Long` предлагают методы `parseByte()`, `parseShort()`, `parseInt()` и `parseLong()` соответственно. Эти методы возвращают значения типа `byte`, `short`, `int` или `long` — эквиваленты числовой строки, с которой они были вызваны (аналогичные методы также предусмотрены в классах `Float` и `Double`).

В следующей программе демонстрируется применение метода `parseInt()`. Программа суммирует список целочисленных значений, введенных пользователем. Для этого программа читает целочисленные значения с помощью метода `readLine()` и использует метод `parseInt()` для преобразования этих строк в их эквиваленты типа `int`.

```
/* Эта программа суммирует список целых чисел, введенных пользователем.
   Она преобразует строковое представление каждого числа в целое,
   используя parseInt()
*/
import java.io.*;
class ParseDemo {
    public static void main(String args[])
    throws IOException
    { // Создать BufferedReader, используя System.in
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        String str;
        int i;
        int sum=0;
        System.out.println("Введите число, 0 — для выхода.");
        do {
            str = br.readLine();
            try {
                i = Integer.parseInt(str);
```

```
            } catch(NumberFormatException e) {
                System.out.println("Неверный формат");
                i = 0;
            }
            sum += i;
            System.out.println("Текущая сумма: " + sum);
        } while(i != 0);
    }
}
```

Чтобы преобразовать число в десятичную строку, используйте версии метода `toString()`, определенные в классах `Byte`, `Short`, `Integer` или `Long`. Классы `Integer` и `Long` также предоставляют методы `toBinaryString()`, `toHexString()` и `toOctalString()`, которые преобразуют значение в бинарную, шестнадцатеричную и восьмеричную строки соответственно.

```
/* Преобразует целое в бинарный, шестнадцатеричный и восьмеричный
   формат
*/
class StringConversions {
    public static void main(String args[]) {
        int num = 19648;
        System.out.println(num + " в бинарной форме: " +
                           Integer.toBinaryString(num));
        System.out.println(num + " в восьмеричной форме: " +
                           Integer.toOctalString(num));
        System.out.println(num + " в шестнадцатеричной форме: " +
                           Integer.toHexString(num));
    }
}
```

Вывод этой программы показан ниже.

```
19648 в бинарной форме: 100110011000000
19648 в восьмеричной форме: 46300
19648 в шестнадцатеричной форме: 4cc0
```

Класс `Character`

Класс `Character` — это простая оболочка для типа `char`. Конструктор этого класса выглядит следующим образом.

```
Character(char символ)
```

Здесь *символ* определяет символ, который будет помещен в оболочку создаваемого объекта класса `Character`. Чтобы получить значение типа `char`, содержащееся в объекте класса `Character`, вызовите метод `charValue()`, показанный ниже.

```
char charValue()
```

Этот метод вернет символ.

В классе `Character` определено несколько констант, включая следующие:

`MAX_RADIX`	Максимальное основание
`MIN_RADIX`	Минимальное основание
`MAX_VALUE`	Максимальное значение
`MIN_VALUE`	Минимальное значение
`TYPE`	Объект класса `Class` для типа `char`

Класс Character включает несколько статических методов, которые категоризируют символы и изменяют их регистр. Они описаны в табл. 16.7. В следующем примере демонстрируется применение некоторых этих методов.

```
// Демонстрация применения некоторых методов Is...
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] + " - десятичное число.");
            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] +
                                   " - символ верхнего регистра.");
            if(Character.isLowerCase(a[i]))
                System.out.println(a[i] + " - символ нижнего регистра.");
        }
    }
}
```

Вывод этой программы выглядит так.

```
a - буква.
a - символ нижнего регистра.
b - буква.
b - символ нижнего регистра.
5 - десятичное число.
A - буква.
A - символ верхнего регистра.
  - пробельный символ.
```

В классе Character определено два метода, forDigit() и digit(), обеспечивающие преобразование между целочисленными значениями и цифрами, которые они представляют. Вот их синтаксис.

```
static char forDigit(int число, int основание)
static int digit(char цифра, int основание)
```

Метод forDigit() возвращает цифровое представление символа, связанного со значением *число*. Основание преобразуемого числа определяет *основание*. Метод digit() возвращает целочисленное значение, связанное с определенным символом (который должен быть цифрой) согласно определенному основанию. (Есть еще одна форма метода digit(), получающая кодовую точку. Более подробно кодовая точка обсуждается в следующем разделе.)

В классе Character определен также метод compareTo(), имеющий следующую форму.

```
int compareTo(Character c)
```

Метод возвращает нуль, если вызывающий объект и *c* имеют одинаковое значение. Он возвращает отрицательное значение, если значение вызывающего объекта ниже. В противном случае он возвращает положительное значение.

Таблица 16.7. Различные методы класса `Character`

Метод	Описание
`static boolean isDefined(char` *символ*`)`	Возвращает значение `true`, если *символ* определен в Unicode. В противном случае возвращает значение `false`
`static boolean isDigit(char` *символ*`)`	Возвращает значение `true`, если *символ* является десятичной цифрой. В противном случае возвращает значение `false`
`static boolean isIdentifierIgnorable(char` *символ*`)`	Возвращает значение `true`, если *символ* должен быть проигнорирован в идентификаторе. В противном случае возвращает значение `false`
`static boolean isISOControl(char` *символ*`)`	Возвращает значение `true`, если *символ* является управляющим символом ISO. В противном случае возвращает значение `false`
`static boolean isJavaIdentifierPart(char` *символ*`)`	Возвращает значение `true`, если *символ* может быть частью идентификатора Java. В противном случае возвращает значение `false`
`static boolean isJavaIdentifierStart(char` *символ*`)`	Возвращает значение `true`, если *символ* может быть первым символом идентификатора Java. В противном случае возвращает значение `false`
`static boolean isLetter(char` *символ*`)`	Возвращает значение `true`, если *символ* — буква. В противном случае возвращает значение `false`
`static boolean isLetterOrDigit(char` *символ*`)`	Возвращает значение `true`, если *символ* — буква или цифра. В противном случае возвращает значение `false`
`static boolean isLowerCase(char` *символ*`)`	Возвращает значение `true`, если *символ* — буква в нижнем регистре. В противном случае возвращает значение `false`
`static boolean isMirrored(char` *символ*`)`	Возвращает значение `true`, если *символ* — зеркальный символ Unicode. Зеркальным называют символ, зарезервированный для текстов, отображаемых справа налево
`static boolean isSpaceChar(char` *символ*`)`	Возвращает значение `true`, если *символ* — пробельный символ Unicode. В противном случае возвращает значение `false`
`static boolean isTitleCase(char` *символ*`)`	Возвращает значение `true`, если *символ* — титульный символ. В противном случае возвращает значение `false`
`static boolean isUnicodeIdentifierPart(char` *символ*`)`	Возвращает значение `true`, если *символ* допустим в качестве части идентификатора Unicode (кроме первого символа). В противном случае возвращает значение `false`

Окончание табл. 16.7

Метод	Описание
`static boolean isUnicodeIdentifierStart(char` *символ*`)`	Возвращает значение `true`, если *символ* допустим в качестве первого символа идентификатора Unicode. В противном случае возвращает значение `false`
`static boolean isUpperCase(char` *символ*`)`	Возвращает значение `true`, если *символ* — символ верхнего регистра. В противном случае возвращает значение `false`
`static boolean isWhitespace(char` *символ*`)`	Возвращает значение `true`, если *символ* — пробельный символ. В противном случае возвращает значение `false`
`static char toLowerCase(char` *символ*`)`	Возвращает эквивалент *символ* в нижнем регистре
`static char toTitleCase(char` *символ*`)`	Возвращает эквивалент *символ* в титульном регистре
`static char toUpperCase(char` *символ*`)`	Возвращает эквивалент *символ* в верхнем регистре

В классе `Character` определено два метода — `forDigits()` и `digit()`, которые позволяют выполнять преобразования между целочисленными значениями и представляющими их цифрами. Выглядят они следующим образом.

```
static char forDigit(int число, int основание)
static int digit(char цифра, int основание)
```

Метод `forDigit()` возвращает десятичную цифру, ассоциированную со значением *число*. Основание для преобразования задается параметром *основание*. Метод `digit()` возвращает целое число, ассоциированное с указанным символом (предполагается, цифрой) в соответствии с заданным основанием.

Еще один метод, определенный в классе `Character`, — это метод `compareTo()`, который имеет следующую форму.

```
int compareTo(Character c)
```

Он возвращает нуль, если вызывающий объект и *c* эквивалентны, и отрицательное значение — если вызывающий объект содержит меньшее значение. В противном случае возвращает положительное значение.

Класс `Character` содержит метод по имени `getDirectionality()`, который может быть использован для определения направления символа. Несколько констант добавлены для описания направления написания символа. Большинство программ в этом не нуждается.

Класс `Character` также переопределяет методы `equals()` и `hashCode()`.

Два других ориентированных на символы класса — это класс `Character.Subset`, используемый для описания подмножества Unicode, и класс `Character.UnicodeBlock`, содержащий блоки символов Unicode.

Дополнения к классу `Character` для поддержки кодовых точек Unicode

В последнее время в классе `Character` появились существенные дополнения. Начиная с JDK 5 класс `Character` обеспечивает поддержку 32-битовых символов Unicode. В прошлом все символы Unicode состояли из 16 бит, что равно размеру

char (и размеру значения, инкапсулированного в классе Character), поскольку эти символы находятся в диапазоне от 0 до FFFF. Однако набор символов Unicode был расширен, и понадобилось более 16 бит. Теперь символы расположены в диапазоне от 0 до 10FFFF.

Появилось три важных термина. *Кодова́я то́чка* (code point) — это символ в диапазоне от 0 до 10FFFF. Символы, имеющие код свыше FFFF, называются *дополнительными символами* (supplemental character). *Базовая многоязыковая плоскость* (Basic Multilingual Plane — BMP) — это символы от 0 до FFFF.

Расширение набора символов Unicode создает фундаментальные проблемы для Java. Поскольку дополнительные символы имеют значение, большее, чем умещается в тип char, для их поддержки требуются дополнительные средства. В Java эта проблема решается двумя способами. Во-первых, язык Java использует две переменные типа char для представления дополнительных символов. Первая из них называется *старшей сигнатурой* (high signature), а вторая — *младшей сигнатурой* (low signature). Для трансляции между кодовыми точками и дополнительными символами предусмотрены новые методы, такие как codePointAt().

Во-вторых, язык Java переопределяет некоторые из ранее существовавших методов класса Character. Перегруженные формы используют данные типа int вместо char. Поскольку тип int достаточно велик, чтобы уместить любой символ как одно значение, его можно использовать для хранения любого символа. Например, все методы из табл. 16.7 имеют перегруженные формы, которые оперируют типом int. Вот примеры.

```
static boolean isDigit(int кодоваяточка)
static boolean isLetter(int кодоваяточка)
static int toLowerCase(int кодоваяточка)
```

В дополнение к методам, перегруженным для работы с кодовыми точками, в классе Character также добавлены методы, которые предлагают дополнительную поддержку кодовых точек. Их примеры можно найти в табл. 16.8.

Таблица 16.8. Примеры методов, которые обеспечивают поддержку 32-битовых кодовых точек Unicode

Метод	Описание
static int charCount(int *кодоваяточка*)	Возвращает 1, если *кодоваяточка* может быть представлена одной переменной типа char. Возвращает 2, если требуется две
static int codePointAt(CharSequence *символы*, int *позиция*)	Возвращает кодовую точку, находящуюся в позиции *позиция*
static int codePointAt(char *символы*[], int *позиция*)	Возвращает кодовую точку, находящуюся в позиции *позиция*
static int codePointBefore(CharSequence *символы*, int *позиция*)	Возвращает кодовую точку, находящуюся в позиции, предшествующей позиции *позиция*
static int codePointBefore(char *символы*[], int *позиция*)	Возвращает кодовую точку, находящуюся в позиции, предшествующей позиции *позиция*
static boolean isBmpCodePoint(int *кодоваяточка*)	Возвращает значение true, если *кодоваяточка* относится к базовой многоязыковой плоскости, и значение false — в противном случае. (Добавлено в JDK 7)

Окончание табл. 16.8

Метод	Описание
`static boolean isHighSurrogate(char` *символ*`)`	Возвращает значение `true`, если *символ* представляет корректный символ верхней сигнатуры
`static boolean isLowSurrogate(char` *символ*`)`	Возвращает значение `true`, если *символ* представляет корректный символ нижней сигнатуры
`static boolean isSupplementaryCodePoint(int` *кодоваяточка*`)`	Возвращает значение `true`, если *символ* представляет собой дополнительный символ
`static boolean isSurrogatePair(char` *верхнСигн*`, char` *нижнСигн*`)`	Возвращает значение `true`, если *верхнСигн* и *нижнСигн* формируют корректную суррогатную пару
`static boolean isValidCodePoint(int` *кодоваяточка*`)`	Возвращает значение `true`, если *кодоваяточка* представляет корректную кодовую точку
`static char[] toChars(int` *кодоваяточка*`)`	Преобразует кодовую точку в *кодоваяточка* в эквивалент типа `char`, который может потребовать двух переменных типа `char`. Возвращает массив, содержащий результат
`static int toChars(int` *кодоваяточка*`, char` *цель*`[], int` *позиция*`)`	Преобразует кодовую точку *кодоваяточка* в эквивалент типа `char`, сохраняя результат в *цель*, начиная с позиции *позиция*. Возвращает 1, если *кодоваяточка* может быть представлена одним символом *символ*. В противном случае возвращает 2
`static int toCodePoint(char` *верхнСигн*`, char` *нижнСигн*`)`	Преобразует *верхнСигн* и *нижнСигн* в эквивалентную кодовую точку

Класс `Boolean`

Класс `Boolean` — это очень тонкая оболочка вокруг значений типа `boolean`, что удобно, в основном, тогда, когда вы хотите передавать значения типа `boolean` по ссылке. Этот класс содержит константы `TRUE` и `FALSE`, определяющие объекты класса `Boolean`, которые соответствуют истинному и ложному значениям. Класс `Boolean` определяет также поле `TYPE`, являющееся объектом класса `Class` для типа `boolean`. В классе `Boolean` определены следующие конструкторы.

```
Boolean(boolean логичЗначение)
Boolean(String логичСтрока)
```

В первой версии значением *логичЗначение* должно быть либо значение `true`, либо значение `false`. Во второй версии конструктора, если *логичСтрока* содержит строку `"true"` (в верхнем или нижнем регистре), то новый объект класса `Boolean` будет содержать значение `true`. В противном случае — значение `false`.

В классе `Boolean` определены методы, перечисленные в табл. 16.9.

Таблица 16.9. Методы класса `Boolean`

Метод	Описание
`boolean booleanValue()`	Возвращает эквивалент типа `boolean`
`static int compare(boolean b1, boolean b2)`	Возвращает значение нуль, если *b1* и *b2* содержат одинаковое значение. Возвращает положительное значение, если *b1* — значение `true` и *b2* — значение `false`, а в противном случае возвращает отрицательное значение. (Добавлено в JDK 7)
`int compareTo(Boolean b)`	Возвращает нуль, если вызывающий объект и *b* содержат одинаковые значения. Возвращает положительное значение, если вызывающий объект равен значению `true`, а `b` — значению `false`. В противном случае возвращает отрицательное значение
`boolean equals(Object` *объектBool*`)`	Возвращает значение `true`, если вызывающий объект эквивалентен *объектBool*. В противном случае возвращает значение `false`
`static boolean getBoolean(String` *имяСвойства*`)`	Возвращает значение `true`, если системное свойство, указанное в параметре *имяСвойства*, содержит значение `true`. В противном случае возвращает значение `false`
`int hashCode()`	Возвращает хеш-код вызывающего объекта
`static boolean parseBoolean(String` *строка*`)`	Возвращает значение `true`, если *строка* содержит строку `"true"`. Регистр символов не важен. В противном случае возвращает значение `false`
`String toString()`	Возвращает строковый эквивалент вызывающего объекта
`String toString(boolean` *логичЗначение*`)`	Возвращает строковый эквивалент *логичЗначение*
`static boolean valueOf(boolean` *логичЗначение*`)`	Возвращает логический эквивалент *логичЗначение*
`static boolean valueOf(String` *логичСтрока*`)`	Возвращает значение `true`, если *логичСтрока* содержит строку `"true"` (в верхнем или нижнем регистре). В противном случае возвращает значение `false`

Класс `Void`

Этот класс имеет одно поле `TYPE`, которое содержит ссылку на объект класса `Class` для типа `void`. Экземпляры этого класса не создаются.

Класс `Process`

Абстрактный класс `Process` инкапсулирует *процесс*, т.е. выполняющуюся программу. Он используется, в основном, как суперкласс для класса объектов, созданных методом `exec()` класса `Runtime` или методом `start()` класса `ProcessBuilder`. Класс `Process` содержит абстрактные методы, описанные в табл. 16.10.

Таблица 16.10. Методы класса `Process` (все абстрактные)

Метод	Описание
`void destroy()`	Прерывает процесс
`int extValue()`	Возвращает код завершения процесса
`InputStream getErrorStream()`	Возвращает входной поток, который читает ввод из выходного потока `err` процесса
`InputStream getOutputStream()`	Возвращает входной поток, который читает ввод из выходного потока `out` процесса
`OutputStream getOutputStream()`	Возвращает выходной поток, который записывает вывод во входной поток `in` процесса
`Int waitFor() throws InterruptedException`	Возвращает код завершения процесса. Этот метод не возвращает управление до тех пор, пока процесс, для которого он вызван, не завершится

Класс `Runtime`

Этот класс инкапсулирует среду времени выполнения. Создать объект класса `Runtime` невозможно. Однако можно получить ссылку на текущий объект класса `Runtime`, вызвав статический метод `Runtime.getRuntime()`. Получив ссылку на текущий объект класса `Runtime`, вы можете вызвать несколько методов, контролирующих состояние и поведение виртуальной машины Java (Java Virtual Machine – JVM). Аплеты и другой не заслуживающий доверия код не могут вызвать ни одного метода класса `Runtime` без передачи исключения `SecurityException`. Наиболее часто используемые методы класса `Runtime` перечислены в табл. 16.11.

Таблица 16.11. Примеры методов, определенных в классе `Runtime`

Метод	Описание
`void addShutdownHook(Thread` *поток*`)`	Регистрирует *поток* как поток, который должен быть запущен при останове виртуальной машины Java
`Process exec(String` *имяПрограммы*`) throws IOException`	Выполняет программу, указанную в параметре *имяПрограммы*, как отдельный процесс. Возвращается объект класса `Process`, описывающий новый процесс
`Process exec(String progName, String` *окружение*`[]) throws IOException`	Выполняет программу, указанную в параметре *имяПрограммы*, как отдельный процесс в окружении, определенном параметром *окружение*. Возвращается объект класса `Process`, описывающий новый процесс
`Process exec(String` *массивКомСтроки*`[]) throws IOException`	Выполняет командную строку, переданную в параметре *массивКомСтроки*, как отдельный процесс. Возвращается объект класса `Process`, описывающий новый процесс
`Process exec(String` *массивКомСтроки*`[], String` *окружение*`[]) throws IOException`	Выполняет командную строку, переданную в параметре *массивКомСтроки*, как отдельный процесс в окружении, описанном параметром *окружение*. Возвращается объект класса `Process`, описывающий новый процесс
`void exit(int` *кодВыхода*`)`	Прерывает выполнение и возвращает значение *кодВыхода* родительскому процессу. По соглашению 0 означает нормальное завершение. Все другие значения символизируют различные типы ошибок

Окончание табл. 16.11

Метод	Описание
`long freeMemory()`	Возвращает приблизительное количество байтов свободной памяти, доступной системе времени выполнения Java
`void gc()`	Инициирует сбор "мусора"
`static Runtime getRuntime()`	Возвращает текущий объект класса `Runtime`
`void halt(int` *код*`)`	Немедленно прерывает работу виртуальной машины Java. Никакие завершающие потоки или финализация не выполняются. Вызывающему процессу возвращается значение *код*
`void load(String` *имяФайлаБиблиотеки*`)`	Загружает динамическую библиотеку, файл которой задан параметром *имяФайлаБиблиотеки*, где должен быть указан полный путь к нему
`void loadLibrary(String` *имяБиблиотеки*`)`	Загружает динамическую библиотеку, имя которой ассоциируется с *имяБиблиотеки*
`boolean removeShutdownHook(Thread` *поток*`)`	Удаляет *поток* из списка потоков, подлежащих запуску при останове виртуальной машины Java. Возвращает значение `true` в случае успеха, т.е. если поток удален
`void runFinalization()`	Инициирует вызовы методов `finalize()` для неиспользованных, но еще не возвращенных объектов
`long totalMemory()`	Возвращает общее количество байтов памяти, доступной программе
`void traceInstructions(boolean` *трассировкаВкл*`)`	Включает и отключает трассировку инструкций в зависимости от значения *трассировкаВкл*. Если *трассировкаВкл* содержит значение `true`, то трассировка включается, а если значение `false` – отключается
`void traceMethodCalls(boolean` *трассировкаВкл*`)`	Включает и отключает трассировку вызовов методов в зависимости от значения *трассировкаВкл*. Если *трассировкаВкл* содержит значение `true`, то трассировка включается, а если значение `false` – отключается

Давайте взглянем на два наиболее популярных способа применения класса `Runtime`: управление памятью и выполнение дополнительных процессов.

Управление памятью

Несмотря на то что язык Java предлагает автоматический сбор "мусора", иногда вы хотите узнать объем распределяемой памяти, занятой объектами и свободной. Вы можете использовать эту информацию, например, для проверки эффективности своего кода или выяснить, сколько еще объектов определенного типа может быть инициализировано. Для получения этих значений служат методы `totalMemory()` и `freeMemory()`.

Как упоминалось в части I, сборщик "мусора" Java запускается периодически для утилизации неиспользуемых объектов. Однако иногда может потребоваться собрать отброшенные объекты до того, как сборщик "мусора" будет запущен в очередной раз. Вы можете запускать его по требованию, вызывая метод `gc()`. Неплохо попробовать запустить метод `gc()` и вслед за ним метод `freeMemory()`,

чтобы получить представление об использовании памяти. Далее выполняйте свой код и снова вызывайте метод `freeMemory()`, чтобы увидеть, сколько памяти он резервирует. В следующей программе иллюстрируется описанная идея.

```
/// Демонстрация применения totalMemory(), freeMemory() и gc().
class MemoryDemo {
    public static void main(String args[]) {
        Runtime r = Runtime.getRuntime();
        long mem1, mem2;
        Integer someints[] = new Integer[1000];

        System.out.println("Всего памяти: " + r.totalMemory());
        mem1 = r.freeMemory();

        System.out.println("Свободной памяти вначале: " + mem1);
        r.gc();
        mem1 = r.freeMemory();
        System.out.println("Свободной памяти после сбора "мусора": " +
                           mem1);

        for(int i=0; i<1000; i++)
            someints[i] = new Integer(i); // Распределить Integer
        mem2 = r.freeMemory();
        System.out.println("Свободной памяти после распределения: " +
                           mem2);
        System.out.println("Использовано памяти для распределения: " +
                            (mem1-mem2));

        // отбросить Integers
        for(int i=0; i<1000; i++) someints[i] = null;

        r.gc(); // запуск сборщика "мусора"

        mem2 = r.freeMemory();
        System.out.println("Свободной памяти после сбора" +
                           " отброшенных Integer: " + mem2);
    }
}
```

Пример вывода этой программы показан здесь (конечно, ваши реальные результаты могут отличаться).

```
Всего памяти: 1048568
Свободной памяти вначале: 751392
Свободной памяти после сборки "мусора": 841424
Свободной памяти после распределения: 824000
Использовано памяти для распределения: 17424
Свободной памяти после сборки отброшенных Integer: 842640
```

Выполнение других программ

В безопасных средах вы можете использовать язык Java для выполнения других тяжеловесных процессов (т.е. программ) в многозадачной операционной системе. Некоторые формы метода `exec()` позволяют указывать программу, которую вы хотите выполнить, а также передать ей входные параметры. Метод `exec()` возвращает объект класса `Process`, который затем может использоваться для управления взаимодействием вашей программы Java с этим вновь запущенным процессом. Поскольку Java может функционировать на множестве платформ в средах разнообразных операционных систем, метод `exec()` сильно зависит от среды.

В следующем примере используется метод exec() для запуска приложения Блокнот (Notepad) – простого текстового редактора Windows. Очевидно, что этот пример должен выполняться на платформе операционной системы Windows.

```
// Демонстрация 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 запуска notepad.");
        }
    }
}
```

Существует несколько альтернативных форм метода exec(), но форма, показанная в этом примере, используется наиболее часто. Объектом класса Process, возвращенным методом exec(), можно манипулировать методами класса Process после того, как программа запущена. Вы можете удалить процесс методом destroy(). Метод waitFor() заставит вашу программу ожидать завершения процесса. Метод exitValue() возвратит значение, которое вернет процесс по завершении. Обычно это будет 0, если никаких проблем не возникнет. Ниже представлен предыдущий пример метода exec(), модифицированный так, чтобы он ожидал завершения запущенного процесса.

```
// Ожидает завершения 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 запуска notepad.");
        }

        System.out.println("Notepad возвратил " + p.exitValue());
    }
}
```

Пока выполняется процесс, вы можете писать в его стандартный ввод и читать его стандартный вывод. Методы getOutputStream() и getInputStream() возвращают дескрипторы стандартных потоков in и out процесса. (Ввод-вывод детально рассматривается в главе 19.)

Класс ProcessBuilder

Класс ProcessBuilder обеспечивает другой способ запуска процессов и управления ими (т.е. программами). Как объяснялось ранее, все процессы представляются классом Process, и процесс может быть запущен методом Runtime.exec(). Класс ProcessBuilder предлагает более развитые средства управления процессами. Например, вы можете установить текущий рабочий каталог и изменить параметры окружения. В классе ProcessBuilder определены следующие конструкторы.

```
ProcessBuilder(List<String> аргументы)
ProccessBuilder(String ... аргументы)
```

Здесь *аргументы* — список аргументов, указывающих имя программы, которую нужно запустить, со всеми необходимыми аргументами командной строки. В первом конструкторе аргументы передаются через список класса `List`. Во втором они указываются через параметр с переменным количеством аргументов. В табл. 16.12 описаны методы, определенные в классе `ProcessBuilder`.

Таблица 16.12. Методы, определенные в классе `ProcessBuilder`

Метод	Описание
`List<String> command()`	Возвращает ссылку на объект класса `List`, которая содержит имя программы и ее аргументы. Изменения в этом списке относятся к вызванному объекту
`ProcessBuilder command(List<String> аргументы)`	Устанавливает имя программы и ее аргументы через параметр *аргументы*. Изменения в этом списке относятся к вызванному объекту. Возвращает ссылку на вызванный объект
`ProcessBuilder command(String ... аргументы)`	Устанавливает имя программы и ее аргументы через параметр *аргументы*. Возвращает ссылку на вызванный объект
`File directory()`	Возвращает текущий рабочий каталог вызванного объекта. Это может быть значение `null`, если каталог тот же, что и у программы Java, которая запустила процесс
`ProcessBuilder directory(File каталог)`	Устанавливает текущий каталог для вызванного объекта. Возвращает ссылку на вызванный объект
`Map<String, String> environment()`	Возвращает переменные окружения, ассоциированные с вызванным объектом, в виде пар "ключ-значение"
`ProcessBuilder inheritIO()`	Заставляет вызванный процесс использовать те же исходный и целевой объекты для стандартных потоков ввода-вывода, что и вызывающий процесс. (Добавлено в JDK 7)
`ProcessBuilder.Redirect redirectError()`	Возвращает целевой объект для стандартной ошибки как объект `ProcessBuilder.Redirect`. (Добавлено в JDK 7)
`ProcessBuilder redirectError(File f)`	Устанавливает определенный файл как целевой объект для стандартной ошибки. Возвращает ссылку вызывающему объекту. (Добавлено в JDK 7)
`ProcessBuilder redirectError(ProcessBuilder.Redirect цель)`	Устанавливает как целевой объект для стандартной ошибки файл, определенный параметром *цель*. Возвращает ссылку вызывающему объекту. (Добавлено в JDK 7)
`boolean redirectErrorStream()`	Возвращает значение `true`, если стандартный поток ошибок перенаправлен на стандартный выходной поток. Возвращает значение `false`, если потоки различны
`ProcessBuilder redirectErrorStream(boolean объединить)`	Если параметр *объединить* содержит значение `true`, то стандартный поток ошибок перенаправляется на стандартный вывод. Если параметр *объединить* содержит значение `false`, то потоки разделены, что является состоянием по умолчанию. Возвращает ссылку вызывающему объекту

Окончание табл. 16.12

Метод	Описание
`ProcessBuilder.Redirect redirectInput()`	Возвращает источник для стандартного ввода как объект `ProcessBuilder.Redirect`. (Добавлено в JDK 7)
`ProcessBuilder redirectInput(File f)`	Устанавливает источник для стандартного ввода, как определенный файл. Возвращает ссылку вызывающему объекту. (Добавлено в JDK 7)
`ProcessBuilder redirectInput(ProcessBuilder.Redirect` *источник*`)`	Устанавливает источник для стандартного ввода как определено параметром *источник*. Возвращает ссылку вызывающему объекту. (Добавлено в JDK 7)
`ProcessBuilder.Redirect redirectOutput()`	Возвращает целевой объект для стандартного вывода как объект `ProcessBuilder.Redirect`. (Добавлено в JDK 7)
`ProcessBuilder redirectOutput(File f)`	Устанавливает определенный файл как цель для стандартного устройства вывода. Возвращает ссылку вызывающему объекту. (Добавлено в JDK 7)
`ProcessBuilder redirectOutput(ProcessBuilder.Redirect` *цель*`)`	Устанавливает цель для стандартного устройства вывода, как определено параметром *цель*. Возвращает ссылку вызывающему объекту. (Добавлено в JDK 7)
`Process start() throws IOException`	Запускает процесс, указанный в вызванном объекте. Другими словами, запускает заданную программу

Обратите в табл. 16.12 внимание на то, что комплект JDK 7 добавляет несколько новых методов, которые используют новый класс `ProcessBuilder.Redirect`. Этот абстрактный класс инкапсулирует источник ввода-вывода или целевой объект, связанный с процессом. Кроме того, эти методы позволяют вам переадресовать источник или целевой объект операций ввода-вывода. Например, вызвав метод `to()`, можно переадресовать в файл, вызвав метод `from()`, можно переадресовать из файла, а вызвав метод `appendTo()` — добавить в файл. Объект класса `File`, связанный с файлом, может быть получен при вызове метода `file()`. Эти методы представлены ниже.

```
static ProcessBuilder.Redirect to(File f )
static ProcessBuilder.Redirect from(File f )
static ProcessBuilder.Redirect appendTo(File f )
File file()
```

Класс `ProcessBuilder.Redirect` поддерживает также метод `type()`, который возвращает значение типа перечисления `ProcessBuilder.Redirect.Type`. Это перечисление описывает тип перенаправления. Его значениями являются `APPEND`, `INHERIT`, `PIPE`, `READ` и `WRITE`. В классе определены также константы `INHERIT` и `PIPE`.

Чтобы создать процесс, используя класс `ProcessBuilder`, нужно просто создать экземпляр класса `ProcessBuilder`, указав имя программы и все необходимые аргументы. Чтобы начать выполнение программы, вызовите метод `start()` для этого созданного экземпляра. Ниже показан пример, который запускает приложение Notepad — текстовый редактор Windows. Обратите внимание на то, что в качестве аргумента передается имя файла, который нужно редактировать.

```
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 запуска notepad.");
        }
    }
}
```

Класс System

Этот класс содержит коллекцию статических методов и переменных. Стандартный ввод, вывод и вывод ошибок исполняющей системы Java хранятся в переменных in, out и err. Методы, определенные в классе System, перечислены в табл. 16.13. Многие из них передают исключение SecurityException, если операция не допускается диспетчером безопасности.

Давайте взглянем на некоторые случаи обычного применения класса System.

Таблица 16.13. Методы, определенные в классе System

Метод	Описание
static void arraycopy(Object *источник*, int *начИсточника*, Object *цель*, int *начЦели*, int *размер*)	Копирует массив. Массив, подлежащий копированию, передается в параметре *источник*, а индекс позиции, с которой начинается копирование из *источник*, — в *начИсточника*. Массив, принимающий копию, передается в *цель*, а индекс позиции в нем, куда нужно начать копирование, — в параметре *начЦели*. Параметр *размер* задает количество копируемых элементов
static String clearProperty(String *которая*)	Удаляет переменную окружения, указанную в *которая*. Возвращается предыдущее значение, ассоциированное с *которая*
static Console console()	Возвращает консоль, ассоциированную с JVM. В случае отсутствия у JVM текущей консоли возвращается значение null
static long currentTimeMillis()	Возвращает текущее время в миллисекундах, прошедших с полуночи 1 января 1970 года
static void exit(int *кодВыхода*)	Прерывает выполнение и возвращает значение *кодВыхода* родительскому процессу (обычно операционной системе). По соглашению 0 означает нормальное завершение. Все другие значения обозначают разные варианты ошибок
static void gc()	Инициирует сбор "мусора"
static Map<String, String> getenv()	Возвращает объект интерфейса Map, содержащий текущие переменные окружения и их значения
static String getenv(String *которая*)	Возвращает значение, ассоциированное с переменной окружения, имя которой передано в *которая*
static Properties getProperties()	Возвращает свойства, ассоциированные с системой времени выполнения Java (класс Properties описан в главе 17)
static String getProperty(String *которая*)	Возвращает свойство, ассоциированное с *которая*. Если описанный объект не найден, возвращает значение null

Окончание табл. 16.13

Метод	Описание
`static String getProperty(String которая, String поумолчанию)`	Возвращает свойство, ассоциированное с *которая*. Если описанный объект не найден, возвращается значение *поумолчанию*
`static SecurityManager getSecurityManager()`	Возвращает текущий диспетчер безопасности или значение `null`, если никакого диспетчера не установлено
`static int identityHashCode(Object объект)`	Возвращает идентификационный хеш-код для *объект*
`static Channel inheritedChannel() throws IOException`	Возвращает канал, унаследованный виртуальной машиной Java. Возвращает значение `null`, если никакого канала не унаследовано
`static String lineSeparator()`	Возвращает строку, которая содержит символы разделителя строки
`static void load(String имяФайлаБиблиотеки)`	Загружает динамическую библиотеку, файл которой задан параметром *имяФайлаБиблиотеки*, включая полный путь доступа
`static void loadLibrary(String имяБиблиотеки)`	Загружает динамическую библиотеку, имя которой ассоциировано с *имяБиблиотеки*
`static void mapLibraryName(String библ)`	Возвращает зависящее от платформы имя библиотеки *библ*
`static long nanoTime()`	Получает наиболее точный таймер системы и возвращает его значение в наносекундах, прошедших от какого-то определенного начального момента. Точность таймера не известна
`static void runFinalization()`	Инициирует вызов методов `finalize()` для неиспользуемых, но еще не утилизированных объектов
`static void setErr(PrintStream потокОшибок)`	Устанавливает стандартный поток ошибок `err` в *потокОшибок*
`static void setIn(InputStream вхПоток)`	Устанавливает стандартный входной поток `in` в *вхПоток*
`static void setOut(PrintStream выхПоток)`	Устанавливает стандартный выходной поток `out` в *выхПоток*
`static void setProperties(Properties систСвойства)`	Устанавливает текущие системные свойства в *систСвойства*
`static String setProperty(String которая, String v)`	Присваивает значение *v* свойству по имени *которая*
`static void setSecurity Manager(SecurityManager диспБез)`	Устанавливает диспетчер безопасности *диспБез*

Использование метода currentTimeMills() для измерения времени выполнения программы

Одно применение класса System, которое, возможно, вы сочтете интересным, — это использование метода currentTimeMills() для фиксации времени выполнения различных частей вашей программы. Метод currentTimeMills() возвращает текущее время в миллисекундах, прошедшее с полуночи 1 января 1970 года. Чтобы хронометрировать часть вашей программы, сохраните это значение непосредственно перед началом выполнения данной части. Немедленно после выполнения вызовите метод currentTimeMills() еще раз. Временем выполнения будет разница между конечным и начальным временем. Сказанное демонстрируется в следующей программе.

```
// Измерение времени выполнения программы.
class Elapsed {
    public static void main(String args[]) {
        long start, end;

        System.out.println("Время перебора от 0 до 100 000 000");

        // Время перебора от 0 до 100 000 000

        start = System.currentTimeMillis(); // получить начальное время
        for(long i=0; i < 100000000L; i++) ;
        end = System.currentTimeMillis(); // получить конечное время

        System.out.println("Время выполнения: " + (end-start));
    }
}
```

Пример запуска этой программы (имейте в виду, что ваши результаты могут отличаться).

```
Время перебора от 0 до 100 000 000
Время выполнения: 10
```

Если таймер вашей системы имеет точность уровня наносекунд, вы можете переписать предшествующую программу с использованием метода nanoTime() вместо метода currentTimeMills(). Например, ниже показано ключевое изменение программы, переписанной для использования метода nanoTime().

```
start = System.nanoTime(); // получить начальное время
for(long i=0; i < 100000000L; i++) ;
end = System.nanoTime(); // получить конечное время
```

Использование метода arraycopy()

Метод arraycopy() может применяться для быстрого копирования массива любого типа из одного места в другое. Это намного быстрее, чем эквивалентный цикл, написанный чисто на языке Java. Ниже показан пример двух массивов, копируемых методом arraycopy(). Вначале массив a копируется в массив b. Затем все элементы массива a сдвигаются вниз на единицу. После этого массив b смещается вверх на одну позицию.

```
// 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));
    }
```

Как вы можете видеть из следующего вывода, копировать можно один и тот же источник и получатель в обоих направлениях.

```
a = ABCDEFGHIJ
b = MMMMMMMMM
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCDEFGHI
b = BCDEFGHIJJ
```

Свойства окружения

Во всех случаях доступны следующие свойства.

file.separator	java.specification.version	java.vm.version
java.class.path	java.vendor	line.separator
java.class.version	java.vendor.url	os.arch
java.compiler	java.version	os.name
java.ext.dirs	java.vm.name	os.version
java.home	java.vm.specification.name	path.separator
java.io.tmpdir	java.vm.specification.vendor	user.dir
java.library.path	java.vm.specification.version	user.home
java.specification.name	java.vm.vendor	user.name
java.specification.vendor		

Вы можете получить значения различных переменных окружения, вызвав метод `System.getProperty()`. Например, следующая программа отображает путь к текущему пользовательскому каталогу.

```
class ShowUserDir {
    public static void main(String args[]) {
        System.out.println(System.getProperty("user.dir"));
    }
}
```

Класс `Object`

Как упоминалось в части I, класс `Object` — это суперкласс для всех других классов. В классе `Object` определены методы, перечисленные в табл. 16.14, которые применимы к любому объекту.

Таблица 16.14. Методы, определенные в классе `Object`

Метод	
`Object clone() throws CloneNotSupportedException`	Создает новый объект, который повторяет вызывающий объект
`boolean equals(Object` *объект*`)`	Возвращает значение `true`, если вызывающий объект эквивалентен *объект*
`void finalize() throws Throwable`	Метод `finalize()` по умолчанию. Вызывается перед удалением неиспользуемого объекта
`final Class <?> getClass()`	Получает объект класса `Class`, который описывает вызывающий объект
`int hashCode()`	Возвращает хеш-код, ассоциированный с вызывающим объектом
`final void notify()`	Прерывает выполнение потока, ожидающего вызывающего объекта
`final void notifyAll()`	Прерывает выполнение всех потоков, ожидающих вызывающего объекта
`String toString()`	Возвращает строку, описывающую объект
`final void wait() throws InterruptedException`	Ожидает завершения выполнения другого потока
`final void wait(long` *миллисекунды*`) throws InterruptedException`	Ожидает до указанного количества миллисекунд завершения выполнения другого потока
`final void wait(long` *миллисекунды*`, int` *наносекунды*`) throws InterruptedException`	Ожидает до указанного количества миллисекунд плюс наносекунд до завершения выполнения другого потока

Использование метода `clone()` и интерфейса `Cloneable`

Большинство определенных в классе `Object` методов описывается в других местах настоящей книги. Однако один из них требует особого внимания — метод `clone()`. Этот метод создает дубликат объекта, который его вызвал. Клонироваться могут только те классы, которые реализуют интерфейс `Cloneable`.

Интерфейс `Cloneable` не объявляет членов. Он служит для указания того факта, что класс допускает побитовое копирование объектов (т.е. *клонирование*). Если вы попытаетесь вызвать метод `clone()` для класса, который не реализует интерфейс `Cloneable`, будет передано исключение `CloneNotSupportedException`. Когда создается клон, конструктор клонируемого объекта *не вызывается*. Клон — это просто точная копия оригинала.

Клонирование — потенциально опасное действие, поскольку оно может вызвать нежелательные побочные эффекты. Например, если объект, подвергаемый клонированию, содержит ссылочную переменную по имени `objRef`, в клоне она будет ссылаться на тот же объект, что и переменная `objRef` в оригинале. Если клон выполнит изменения в содержимом объекта, на который ссылается переменная `objRef`, то это также изменит исходный объект. Вот другой пример: если объект открывает поток ввода-вывода, а затем клонируется, то два объекта будут

иметь дело с одним и тем же потоком. Более того, если один из этих объектов закроет поток, второй все еще может пытаться писать в него, что приведет к ошибке. В некоторых случаях вам понадобится переопределить метод `clone()`, определенный в классе `Object`, чтобы справиться с проблемами подобного рода.

Поскольку клонирование может вызывать проблемы, метод `clone()` объявляется внутри класса `Object` как `protected`. Это значит, что он либо может быть вызван из метода, определенного в классе, реализующем интерфейс `Cloneable`, либо должен быть явно переопределен в классе как `public`. Давайте рассмотрим примеры каждого подхода. В следующей программе реализуется интерфейс `Cloneable` и определяется метод `cloneTest()`, который вызывает метод `clone()` в классе `Object`.

```
// Демонстрация применения 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 невозможно.");
            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 метод `clone()` класса `Object` и возвращает результат. Обратите внимание на то, что объект, возвращенный методом `clone()`, должен быть приведен к соответствующему типу (`TestClone`).

В следующем примере метод `clone()` перегружается таким образом, что он может быть вызван извне класса. Чтобы сделать это, спецификатором доступа должен быть `public`, как показано ниже.

```
// Переопределение метода clone().
class TestClone implements Cloneable {
    int a;
    double b;
    // clone() переопределен как public.
    public Object clone() {
        try {
            // вызов clone() из Object.
            return super.clone();
        } catch(CloneNotSupportedException e) {
            System.out.println("Клонирование невозможно.");
            return this;
```

```
        }
    }
}

class CloneDemo2 {
    public static void main(String args[]) {
        TestClone x1 = new TestClone();
        TestClone x2;
        x1.a = 10;
        x1.b = 20.98;
        // здесь clone() вызывается непосредственно.
        x2 = (TestClone) x1.clone();
        System.out.println("x1: " + x1.a + " " + x1.b);
        System.out.println("x2: " + x2.a + " " + x2.b);
    }
}
```

Побочные эффекты от клонирования иногда поначалу трудно обнаружить. Очень легко решить, что класс безопасен для клонирования, когда на самом деле это не так. Вообще говоря, вы не должны реализовать интерфейс `Cloneable` для любого класса без серьезной на то причины.

Класс `Class`

Этот класс инкапсулирует состояние времени выполнения класса или интерфейса. Объекты класса `Class` *создаются автоматически при загрузке класса. Вы* не можете явно объявлять объект класса `Class`. В общем случае вы получаете объект класса `Class`, вызывая метод `getClass()`, определенный в классе `Object`. Класс `Class` — это обобщенный тип, который объявлен, как показано ниже.

```
Class Class<T>
```

Здесь *T* — тип представленного класса или интерфейса. Примеры наиболее часто используемых методов, определенных в классе `Class`, приведены в табл. 16.15.

Таблица 16.15. Примеры методов, определенных в классе `Class`

Метод	Описание
`static Class<?> forName(String` *имя*`)` `throws ClassNotFoundException`	Возвращает объект класса `Class` по его полному имени
`static Class<?> forName(String` *имя*`,` `boolean` *как*`, ClassLoader` *загр*`)` `throws ClassNotFoundException`	Возвращает объект класса `Class` по его полному имени. Объект загружается с помощью загрузчика, указанного в *загр*. Если параметр *как* равен значению `true`, объект инициализируется, в противном случае — нет
`<A extends Annotation>` `A getAnnotation(Class<A>` *типАннотации*`)`	Возвращает объект интерфейса `Annotation`, содержащий аннотацию, ассоциированную с *типАннотации*, для вызывающего объекта
`Annotation[ ] getAnnotations()`	Получает аннотацию, ассоциированную с вызывающим объектом, и сохраняет ее в массиве объектов интерфейса `Annotation`. Возвращает ссылку на массив

Продолжение табл. 16.15

Метод	Описание
`Class<?>[ ] getClasses()`	Возвращает объекты класса `Class` для каждого открытого класса и интерфейса, являющихся членами класса, представленного вызывающим объектом
`ClassLoader getClassLoader()`	Возвращает объект класса `ClassLoader`, который загружал класс или интерфейс
`Constructor<T> getConstructor(Class ... типыПараметров) throws NoSuchMethodException, SecurityException`	Возвращает объект класса `Constructor`, представляющий конструктор класса, предоставленного вызывающим объектом, который имеет типы параметров, указанные в *типыПараметров*
`Constructor<?>[] getConstructors() throws SecurityException`	Получает объекты класса `Constructor` для каждого из открытых конструкторов класса, представленного вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на этот массив
`Annotation[] getDeclaredAnnotations()`	Получает объекты интерфейса `Annotation` для всех аннотаций, объявленных в вызывающем объекте, и сохраняет их в массиве. Возвращает ссылку на этот массив (унаследованные аннотации игнорируются)
`Constructor<?>[] getDeclaredConstructors() throws SecurityException`	Получает объекты класса `Constructor` для каждого из объявленных конструкторов класса, предоставленных вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на этот массив (конструкторы суперклассов игнорируются)
`Field[] getDeclaredFields() throws SecurityException`	Получает объекты класса `Field` для каждого из полей, объявленных в классе или интерфейсе, представленном вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на массив. (Унаследованные поля игнорируются)
`Method[] getDeclaredMethods() throws SecurityException`	Получает объекты класса `Method` для каждого из методов, объявленных в классе или интерфейсе, представленном вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на этот массив. (Унаследованные методы игнорируются)
`Field getField(String имяПоля) throws NoSuchMethodException, SecurityException`	Возвращает объект класса `Field`, который представляет открытое поле, указанное в параметре *имяПоля* для класса или интерфейса, представляющего вызывающий объект
`Field[] getFields() throws SecurityException`	Получает объекты класса `Field` для каждого открытого поля, класса или интерфейса, представляющего вызывающий объект, и сохраняет их в массиве. Возвращает ссылку на этот массив

Окончание табл. 16.15

Метод	Описание
`Class<?>[] getInterfaces()`	При вызове для объекта, представляющего класс или интерфейс, этот метод возвращает массив интерфейсов, реализованных этим классом. Когда вызывается для объекта, представляемого интерфейсом, этот метод возвращает массив интерфейсов, расширяемых данным интерфейсом
`Method getMethod(String` *имяМетода*`, Class<?> ...` *типыПараметров*`) throws NoSuchMethodException, SecurityException`	Возвращает объект класса `Method`, который представляет открытый метод, заданный параметром *имяМетода* с параметрами типов, указанных в параметрах *типыПараметров* в классе или интерфейсе, представляемом вызывающим объектом
`Method[] getMethods() throws SecurityException`	Получает объекты класса `Method` для каждого открытого метода класса или интерфейса, представляемого вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на этот массив
`String getName()`	Возвращает полное имя класса или интерфейса типа, представляемого вызывающим объектом
`ProtectionDomain getProtectionDomain()`	Возвращает домен защиты, связанный с вызывающим объектом
`Class<? super T> getSuperclass()`	Возвращает суперкласс типа, представленного вызывающим объектом. Возвращает значение `null`, если представленный тип – класс `Object` или не класс
`boolean isInterface()`	Возвращает значение `true`, если тип, представленный вызывающим объектом, является интерфейсом. В противном случае возвращает значение `false`
`T newInstance( ) throws IllegalAccessException, InstantiationException`	Создает новый экземпляр (т.е. новый объект), тип которого совпадает с типом представленного вызывающего объекта. Это эквивалентно использованию оператора `new` со стандартным конструктором класса. Возвращается новый объект. Этот метод потерпит неудачу, если представленный тип будет абстрактным, не классом или не будет иметь стандартного конструктора
`String toString()`	Возвращает строковое представление типа, представленного вызывающим объектом или интерфейсом

Методы, определенные в классе `Class`, часто применяются в ситуациях, когда требуется информация об объекте времени выполнения. Как показано в табл. 16.15, вам предоставляются методы, которые позволяют получить дополнительную информацию об определенном классе, такую как его открытые конструкторы, поля и методы. Кроме того, это важно для обеспечения функциональных возможностей Java Beans, которые обсуждаются далее.

В следующей программе демонстрируется применение метода `getClass()` (унаследованного от класса `Object`) и метода `getSuperclass()` (из класса `Class`).

```
// Работа с информацией о типе времени 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("Суперкласс y: " +
        clObj.getName());
    }
}
```

Ниже показан вывод этой программы.

```
x — объект типа: X
y — объект типа: Y
Суперкласс y: X
```

Класс `ClassLoader`

Абстрактный класс `ClassLoader` определяет, как загружаются классы. Ваше приложение может создавать подклассы, расширяющие класс `ClassLoader`, реализуя его методы. Это позволяет загружать классы другими способами, нежели тот, которым выполняется обычная загрузка в системе времени выполнения Java. Однако обычно вы не должны этого делать.

Класс `Math`

Этот класс содержит все функции с плавающей точкой, которые используются в геометрии и тригонометрии, а также некоторые методы общего назначения. В классе `Math` определены две константы типа `double`: `E` (равная приблизительно 2,72) и `PI` (равная примерно 3,14).

Тригонометрические функции

Методы, перечисленные в табл. 16.16, принимают параметр типа `double`, выражающий угол в радианах и возвращающий результат соответствующей тригонометрической функции.

Таблица 16.16. Прямые тригонометрические функции класса `Math`

Метод	Описание
`static double sin(double arg)`	Возвращает синус угла *arg*, переданного в радианах
`static double cos(double arg)`	Возвращает косинус угла *arg*, переданного в радианах
`static double tan(double arg)`	Возвращает тангенс угла *arg*, переданного в радианах

Методы, перечисленные в табл. 16.17, принимают в качестве параметра результат тригонометрической функции и возвращают угол в радианах, который порождает такой результат. Они представляют собой инверсию соответствующих функций из предыдущей таблицы.

Таблица 16.17. Обратные тригонометрические функции класса `Math`

Метод	Описание
`static double asin(double arg)`	Возвращает угол, синус которого равен *arg*
`static double acos(double arg)`	Возвращает угол, косинус которого равен *arg*
`static double atan(double arg)`	Возвращает угол, тангенс которого равен *arg*
`static double atan2(double x, double y)`	Возвращает угол, тангенс которого равен *x/y*

Методы, перечисленные в табл. 16.18, вычисляют гиперболический синус, косинус и тангенс угла.

Таблица 16.18. Гиперболические функции класса `Math`

Метод	Описание
`static double sinh(double arg)`	Возвращает гиперболический синус угла *arg*, переданного в радианах
`static double cosh(double arg)`	Возвращает гиперболический косинус угла *arg*, переданного в радианах
`static double tanh(double arg)`	Возвращает гиперболический тангенс угла *arg*, переданного в радианах

Экспоненциальные функции

В классе `Math` определен набор экспоненциальных методов, которые описаны в табл. 16.19.

Таблица 16.19. Экспоненциальные функции класса `Math`

Метод	Описание
`static double cbrt(double arg)`	Возвращает кубический корень из *arg*
`static double exp(double arg)`	Возвращает экспоненту *arg*

Окончание табл. 16.19

Метод	Описание
`static double expm1(double arg)`	Возвращает экспоненту *arg* -1
`static double log(double arg)`	Возвращает натуральный алгоритм *arg*
`static double log10(double arg)`	Возвращает логарифм по основанию 10 от *arg*
`static double log1p(double arg)`	Возвращает натуральный алгоритм *arg* +1
`static double pow(double y, double x)`	Возвращает *y* в степени *x*, например `pow(2.0, 3.0)` вернет 8.0
`static double scalb(double arg, int показатель)`	Возвращает *arg*×2*показатель*
`static float scalb(float arg, int показатель)`	Возвращает *arg*×2*показатель*
`static double sqrt(double arg)`	Возвращает квадратный корень из *arg*

Функции округления

В классе `Math` определены некоторые методы, которые предназначены для выполнения различных операций округления. Они перечислены в табл. 16.20. Обратите внимание на два метода `ulp()` в конце таблицы. В данном контексте "ulp" означает "units in the last place" (единицы на последнем месте). Это указывает дистанцию между значением и ближайшим большим значением. Это можно использовать для получения точности результата.

Таблица 16.20. Методы округления класса `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 *arg*
`static double floor(double arg)`	Возвращает наибольшее целое число, которое меньше или равно *arg*
`static int max(int x, int y)`	Возвращает большее из двух чисел *x* и *y*
`static long max(long x, long y)`	Возвращает большее из двух чисел *x* и *y*
`static float max(float x, float y)`	Возвращает большее из двух чисел *x* и *y*
`static double max(double x, double y)`	Возвращает большее из двух чисел *x* и *y*
`static int min(int x, int y)`	Возвращает меньшее из двух чисел *x* и *y*
`static long min(long x, long y)`	Возвращает меньшее из двух чисел *x* и *y*
`static float min(float x, float y)`	Возвращает меньшее из двух чисел *x* и *y*
`static double min(double x, double y)`	Возвращает меньшее из двух чисел *x* и *y*
`static double nextAfter(double arg, double направл)`	Начиная со значения *arg* возвращает следующее значение в направлении *направл*. Если *arg*==*направл*, то возвращается *направл*

Окончание табл. 16.20

Метод	Описание
`static float nextAfter(float` *arг*, `double` *направл*)	Начиная со значения *arг* возвращает следующее значение в направлении *направл*. Если *arг*==*направл*, то возвращается *направл*
`static double nextUp(double` *arг*)	Возвращает следующее значение в положительном направлении от *arг*
`static float nextUp(float` *arг*)	Возвращает следующее значение в положительном направлении от *arг*
`static double rint(double` *arг*)	Возвращает ближайшее целое к *arг*
`static int round(float` *arг*)	Возвращает *arг*, округленное вверх до ближайшего `int`
`static long round(double` *arг*)	Возвращает *arг*, округленное вверх до ближайшего `long`
`static float ulp(float` *arг*)	Возвращает `ulp` для *arг*
`static double ulp(double` *arг*)	Возвращает `ulp` для *arг*

Прочие методы класса `Math`

В дополнение к методам, приведенным в таблицах выше, в классе `Math` определено еще несколько методов, которые перечислены в табл. 16.21.

Таблица 16.21. Прочие методы класса `Math`

Метод	Описание
`static double copySign(double` *arг*, `double` *знакарг*)	Возвращает *arг* с тем же знаком, что у *знакарг*
`static float copySign(float` *arг*, `float` *знакарг*)	Возвращает *arг* с тем же знаком, что у *знакарг*
`static int getExponent(double` *arг*)	Возвращает экспоненту по основанию 2, используемую для двоичного представления *arг*
`static int getExponent(float` *arг*)	Возвращает экспоненту по основанию 2, используемую для двоичного представления *arг*
`static double IEEEremainder(double` *делимое*, `double` *делитель*)	Возвращает остаток от деления *делимое*/*делитель*
`static hypot(double` *сторона1*, `double` *сторона2*)	Возвращает длину гипотенузы прямоугольного треугольника по длине двух противоположных сторон
`static double random()`	Возвращает псевдослучайное число между 0 и 1
`static float signum(double` *arг*)	Определяет знак значения. Возвращает 0, если *arг* равен 0, 1 — если *arг* больше 0 и –1 — если *arг* меньше 0
`static float signum(float` *arг*)	Определяет знак значения. Возвращает 0, если *arг* равен 0, 1 — если *arг* больше 0 и –1 — если *arг* меньше 0

Окончание табл. 16.21

Метод	Описание
`static double toDegrees(double угол)`	Преобразует радианы в градусы. Угол, переданный параметром *угол*, должен быть указан в радианах. Возвращается результат в градусах
`static double toRadians(double угол)`	Преобразует градусы в радианы. Угол, переданный параметром *угол*, должен быть указан в градусах. Возвращается результат в радианах

В следующей программе демонстрируется использование методов `toRadians()` и `toDegrees()`.

```
// Демонстрация применения 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.");
    }
}
```

Вывод этой программы показан ниже.

```
120.0 градусов равно 2.0943951023931953 радиан.
1.312 радиан равно 75.17206272116401 радиан.
```

Класс `StrictMath`

Этот класс определяет полный набор математических методов, аналогичный доступному в классе `Math`. Отличие в том, что версия класса `StrictMath` гарантирует точность, идентичную на всех реализациях языка Java, в то время как класс `Math` имеет большую свободу в целях достижения производительности.

Класс `Compiler`

Класс `Compiler` поддерживает создание окружений Java, в которых код виртуальной машины Java компилируется в исполняемый код вместо интерпретируемого. При нормальном программировании не используется.

Классы `Thread`, `ThreadGroup` и интерфейс `Runnable`

Интерфейс `Runnable`, а также классы `Thread` и `ThreadGroup` поддерживают многопоточное программирование. Они рассматриваются ниже.

На заметку! Краткий обзор технологий, используемых для работы с потоками, реализация интерфейса `Runnable` и создание многопоточных программ представлены в главе 11.

Интерфейс Runnable

Этот интерфейс должен быть реализован любым классом, который инициирует отдельный поток выполнения. Интерфейс Runnable определяет только один абстрактный метод по имени run(), который является "точкой входа" потока. Он определен следующим образом.

```
void run()
```

Потоки, которые вы создаете, должны реализовать этот метод.

Класс Thread

Класс Thread создает новый поток выполнения. Он реализует интерфейс Runnable и определяет следующие часто используемые конструкторы.

```
Thread()
Thread(Runnable объектПотока)
Thread(Runnable объектПотока, String имяПотока)
Thread(String имяПотока)
Thread(ThreadGroup группаПотоков, Runnable объектПотока)
Thread(ThreadGroup группаПотоков, Runnable объектПотока, String имяПотока)
Thread(ThreadGroup группаПотоков, String имяПотока)
```

Параметр *объектПотока* — это экземпляр класса, реализующего интерфейс Runnable и определяющий, где начинается выполнение потока. Имя потока задается параметром *имяПотока*. Когда имя не указано, оно создается виртуальной машиной Java. Параметр *группаПотоков* определяет группу потоков, к которой новый поток будет относиться. Когда никакой группы не указано, новый поток относится к той же группе, что и родительский.

В классе Thread определены следующие константы:

```
MAX_PRIORITY
MIN_PRIORITY
NORM_PRIORITY
```

Как и можно было ожидать, эти константы определяют максимальный, минимальный и нормальный уровни приоритета потоков. Методы, определенные в классе Thread, перечислены в табл. 16.22. В ранних версиях языка Java класс Thread включал также методы stop(), suspend() и resume(). Однако, как уже объяснялось в главе 11, они являются нежелательными, поскольку нестабильны. Также нежелательным есть метод countStackFrames(), поскольку он вызывает методы suspend() и destroy() и может привести к взаимной блокировке.

Таблица 16.22. Методы, определенные в классе Thread

Метод	Описание
`static int activeCount()`	Возвращает примерное количество активных потоков в группе, к которой относится данный поток
`void checkAccess()`	Принуждает диспетчер безопасности проверять, что текущий поток способен получать доступ к потоку и/или изменять поток, в котором был вызван метод checkAccess()
`static Thread currentThread()`	Возвращает объект класса Thread, который инкапсулирует поток, вызвавший этот метод
`static void dumpStack()`	Отображает стек вызовов потока

Продолжение табл. 16.22

Метод	Описание
`static int enumerate(Thread потоки[])`	Помещает копию объектов класса `Thread` из текущей группы потоков в массив *потоки*. Возвращается количество потоков
`static Map<Thread, StackTraceElement[]> getAllStackTraces()`	Возвращает объект интерфейса `Map`, который содержит трассировку стека для всех активных потоков в группе. Каждый элемент карты состоит из ключа, который является объектом класса `Thread`, и его значения, которое представляет собой массив элементов класса `StackTraceElement`
`ClassLoader getContextClassLoader()`	Возвращает контекст загрузчика класса, используемый для загрузки классов и ресурсов для текущего потока
`static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()`	Возвращает обработчик неперехваченных исключений по умолчанию
`Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()`	Возвращает обработчик неперехваченных исключений данного потока
`long getID()`	Возвращает идентификатор вызывающего потока
`final String getName()`	Возвращает имя потока
`final int getPriority()`	Возвращает установки приоритета потока
`StackTraceElement[] getStackTrace()`	Возвращает массив, содержащий трассировку стека вызывающего потока
`Thread.State getState()`	Возвращает состояние вызывающего потока
`final ThreadGroup getThreadGroup()`	Возвращает объект класса `ThreadGroup`, членом которого является текущий поток
`static boolean holdsLock(Object объект)`	Возвращает значение `true`, если вызывающий поток владеет блокировкой на объект *объект*. В противном случае возвращает значение `false`
`void interrupt()`	Прерывает поток
`static boolean interrupted()`	Возвращает значение `true`, если вызывающий поток запланирован на прерывание. В противном случае возвращает значение `false`
`final Boolean isAlive()`	Возвращает значение `true`, если вызывающий поток еще активен. В противном случае возвращает значение `false`
`final Boolean isDaemon()`	Возвращает значение `true`, если вызывающий поток является потоком-демоном (одним из потоков исполняющей системы Java). В противном случае возвращает значение `false`
`boolean isInterrupted()`	Возвращает значение `true`, если вызывающий поток прерван. В противном случае возвращает значение `false`

Окончание табл. 16.22

Метод	Описание
`final void join() throws InterruptedException`	Ожидает завершения потока
`final void join(long` *миллисекунды*`) throws InterruptedException`	Ожидает до заданного количества миллисекунд завершения потока, в котором вызван метод
`final void join(long` *миллисекунды*`, int` *наносекунды*`) throws InterruptedException`	Ожидает до заданного количества миллисекунд плюс наносекунд завершения потока, в котором метод вызван
`void run()`	Начинает выполнение потока
`void setContextClassLoader(ClassLoader` *cl*`)`	Устанавливает контекст загрузчика класса, который будет использован вызывающим объектом
`final void setDaemon(boolean` *состояние*`)`	Помечает поток как поток-демон
`static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler` *e*`)`	Устанавливает обработчик неперехваченных прерываний по умолчанию
`void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler` *e*`)`	Устанавливает обработчик неперехваченных прерываний по умолчанию для вызывающего потока
`final void setName(String` *имяПотока*`)`	Устанавливает имя потока в *имяПотока*
`final void setPriority(int` *приоритет*`)`	Устанавливает приоритет потока в *приоритет*
`static void sleep(long` *миллисекунды*`) throws InterruptedException`	Прерывает выполнение потока на заданное количество миллисекунд
`static void sleep(long` *миллисекунды*`, int` *наносекунды*`) throws InterruptedException`	Прерывает выполнение потока на заданное количество миллисекунд плюс наносекунд
`void start()`	Запускает выполнение потока
`String toString()`	Возвращает строковое представление потока
`static void yield()`	Вызывающий поток предлагает уступить ресурс центрального процессора другому потоку

Класс ThreadGroup

Описываемый класс создает группу потоков. В классе `ThreadGroup` определены следующие два конструктора.

```
ThreadGroup(String имяГруппы)
ThreadGroup(ThreadGroup родительскийОбъект, String имяГруппы)
```

Для обеих форм параметр *имяГруппы* указывает имя группы потоков. Первая версия создает новую группу, родителем которой будет текущий поток. Во второй форме родитель группы задается параметром *родительскийОбъект*. Неустаревшие методы, определенные в классе `ThreadGroup`, перечислены в табл. 16.23.

Таблица 16.23. Методы, определенные в классе ThreadGroup

Метод	Описание
`int activeCount()`	Возвращает приблизительное количество активных потоков вызывающей группы (включая данную подгруппу)
`int activeGroupCount()`	Возвращает приблизительное количество активных групп (включая данную подгруппу), для которых вызывающий поток является родителем
`final void checkAccess()`	Вынуждает диспетчер безопасности проверять, может ли вызывающий поток получить доступ к группе, для которой вызван метод `checkAccess()`, и/или изменять ее
`final void destroy()`	Уничтожает группу потоков (и ее дочерние группы), для которой вызывался метод
`int enumerate(Thread` *группа*`[])`	Помещает активные потоки, которые включены в вызывающую группу (включая таковые в подгруппе), в массив *группа*
`int enumerate(Thread` *группа*`[], boolean` *все*`)`	Помещает активные потоки, которые включены в вызывающую группу, в массив *группа*. Если параметр *все* содержит значение `true`, то в массив *группа* помещаются также все подгруппы потока
`int enumerate(ThreadGroup` *группа*`[])`	Помещает активные подгруппы (включая подгруппы подгрупп и т.д.) вызывающей группы в массив *группа*
`int enumerate(ThreadGroup` *группа*`[], boolean` *все*`)`	Помещает активные подгруппы вызывающей группы в массив *группа*. Если параметр *все* содержит значение `true`, то все активные подгруппы всех подгрупп также помещаются в массив *группа*
`final int getMaxPriority()`	Возвращает максимальный приоритет, установленный для группы
`final String getName()`	Возвращает имя группы
`final ThreadGroup getParent()`	Возвращает значение `null`, если вызывающий объект класса `ThreadGroup` не имеет родителя. В противном случае возвращается родитель вызывающего объекта
`final void interrupt()`	Вызывает метод `interrupt()` для всех потоков в группе и любой подгруппе
`final boolean isDaemon()`	Возвращает значение `true`, если текущая группа является группой-демоном. В противном случае возвращается значение `false`
`boolean isDestroyed()`	Возвращает значение `true`, если текущая группа ликвидирована. В противном случае возвращается значение `false`
`void list()`	Отображает информацию о группе
`final boolean parentOf(ThreadGroup` *группа*`)`	Возвращает значение `true`, если вызывающий поток является родителем группы (или самой группой). В противном случае возвращает значение `false`

Окончание табл. 16.23

Метод	Описание
`final void setDaemon(Boolean` *`еслиДемон`*`)`	Если параметр *еслиДемон* содержит значение `true`, то вызывающая группа помечается как группа-демон
`final void setMaxPriority(int` *`приоритет`*`)`	Устанавливает максимальный приоритет для вызывающей группы равным *приоритет*
`String toString()`	Возвращает строковое представление группы
`void uncaughtException(Thread` *`поток`*`, Throwable` *`e`*`)`	Метод вызывается при передаче необработанного исключения

Группы потоков предоставляют удобный способ управления группами потоков как одним целым. В частности, это важно в ситуациях, когда вы хотите приостановить или продолжить выполнение множества взаимосвязанных потоков. Например, предположим, что имеется программа, в которой один набор потоков используется для печати документов, другой – для отображения документа на экране, а еще один сохраняет документ в дисковом файле. Если печать прерывается, потребуется прервать все потоки, имеющие отношение к печати. Сказанное иллюстрируется в следующей программе, которая создает две группы процессов, по два процесса в каждой.

```
// Демонстрация применения групп потоков.
class NewThread extends Thread {
    boolean suspendFlag;

    NewThread(String threadname, ThreadGroup tgOb) {
        super(tgOb, threadname);
        System.out.println("Новый поток: " + this);
        suspendFlag = false;
        start(); // Запуск потока
    }

    // Точка входа 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() + " завершается.");
    }
    synchronized void mysuspend() {
        suspendFlag = true;
    }
    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}
```

```
class ThreadGroupDemo {
    public static void main(String args[]) {
        ThreadGroup groupA = new ThreadGroup("Группа A");
        ThreadGroup groupB = new ThreadGroup("Группа B");

        NewThread ob1 = new NewThread("Один", groupA);
        NewThread ob2 = new NewThread("Два", groupA);
        NewThread ob3 = new NewThread("Три", groupB);
        NewThread ob4 = new NewThread("Четыре", groupB);

        System.out.println("\nВывод из list():");
        groupA.list();
        groupB.list();
        System.out.println();

        System.out.println("Прерывается группа A");
        Thread tga[] = new Thread[groupA.activeCount()];
        groupA.enumerate(tga); // получить потоки группы

        for(int i = 0; i < tga.length; i++) {
            ((NewThread)tga[i]).mysuspend(); // приостановить каждый
                                             // поток
        }

        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 все потоки в
                                            // группе
        }

        // Ожидать завершения потоков
        try {
            System.out.println("Ожидание останова потоков.");
            ob1.join();
            ob2.join();
            ob3.join();
            ob4.join();
        } catch (Exception e) {
            System.out.println("Исключение в основном потоке");
        }
        System.out.println("Основной поток завершен.");
    }
}
```

Пример вывода этой программы показан ниже (точный вывод, который вы увидите, может несколько отличаться).

```
Новый поток: Thread[Один,5,Группа A]
Новый поток: Thread[Два,5,Группа A]
Новый поток: Thread[Три,5,Группа B]
Новый поток: Thread[Четыре,5,Группа B]
Вывод из list():
java.lang.ThreadGroup[name=Group A,maxpri=10]
Thread[Один,5,Группа A]
```

```
Thread[Два,5,Группа A]
java.lang.ThreadGroup[name=Group B,maxpri=10]
Thread[Три,5,Группа B]
Thread[Четыре,5,Группа B]
Прерывается группа A
Три: 5
Четыре: 5
Три: 4
Четыре: 4
Три: 3
Четыре: 3
Три: 2
Четыре: 2
Возобновление группы A
Ожидание останова потоков.
Один: 5
Два: 5
Три: 1
Четыре: 1
Один: 4
Два: 4
Три завершается.
Четыре завершается.
Один: 3
Два: 3
Один: 2
Два: 2
Один: 1
Два: 1
Один завершается.
Два завершается.
Основной поток завершен.
```

Обратите внимание на то, что внутри программы группа A приостанавливается на четыре секунды. Как подтверждает вывод программы, приостанавливается выполнение потоков "Один" и "Два", но потоки "Три" и "Четыре" продолжают выполняться. По истечении четырех секунд выполнение потоков "Один" и "Два" возобновляется. Обратите также внимание на то, как останавливается и возобновляется выполнение группы A. Сначала потоки из группы A извлекаются вызовом метода `enumerate()` для этой группы. Затем каждый поток приостанавливается в процессе итерации по результирующему массиву. Чтобы продолжить выполнение потоков в группе A, опять осуществляется проход по списку потоков, и каждый поток запускается для продолжения работы. И последний момент: в этом примере применяется рекомендованный подход для приостановки и возобновления потоков. Это не касается устаревших (не рекомендованных) методов `suspend()` и `resume()`.

Классы ThreadLocal и InheritableThreadLocal

В пакете `java.lang` определены два дополнительных класса, имеющих отношение к потокам.

- Класс `ThreadLocal` используется для создания локальных переменных потоков. Каждый поток будет иметь собственную копию локальной переменной потока.

- Класс `InheritableThreadLocal` создает локальные переменные потоков, которые могут наследоваться.

Класс `Package`

Этот класс инкапсулирует данные о версии, ассоциированные с пакетом. Информация о версии пакета приобретает особую ценность из-за профилирования пакетов, а также потому, что программы Java могут нуждаться в знании о том, какая версия пакета доступна. Методы, определенные в классе `Package`, перечислены в табл. 16.24. В следующей программе демонстрируется использование класса `Package`, отображающего список пакетов, с которыми имеет дело программа в данный момент.

```
// Демонстрация применения 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()
            );
    }
}
```

Таблица 16.24. Методы, определенные в классе `Package`

Метод	Описание
`<A extends Annotation> A getAnnotation(Class<A>` *типАннотации*`)`	Возвращает объект интерфейса `Annotation`, который содержит аннотацию, ассоциированную с *типАннотации*, для вызывающего объекта
`Annotation[ ] getAnnotations()`	Возвращает все аннотации, ассоциированные с вызывающим объектом в массиве объектов интерфейса `Annotation`. Возвращает ссылку на этот массив
`Annotation[ ] getDeclaredAnnotations()`	Возвращает объект интерфейса `Annotation` для всех аннотаций, объявленных в вызывающем объекте (унаследованные аннотации игнорируются)
`String getImplementationTitle()`	Возвращает заголовок вызывающего пакета
`String getImplementationVendor()`	Возвращает имя реализатора вызывающего пакета
`String getImplementationVersion()`	Возвращает номер версии вызывающего пакета
`String getName()`	Возвращает имя вызывающего пакета
`static Package getPackage(String` *имяПакета*`)`	Возвращает объект класса `Package` по имени, указанном в параметре *имяПакета*

Окончание табл. 16.24

Метод	Описание
`static Package[] getPackages()`	Возвращает все пакеты, о которых осведомлена текущая выполняющаяся программа
`String getSpecificationTitle()`	Возвращает титул спецификации вызывающего пакета
`String getSpecificationVendor()`	Возвращает имя владельца из спецификации вызывающего пакета
`String getSpecificationVersion()`	Возвращает номер версии спецификации вызывающего пакета
`int hashCode()`	Возвращает хеш-код вызывающего пакета
`boolean isAnnotationPresent(Class<? extends Annotation> аннотация)`	Возвращает значение `true`, если аннотация, описанная параметром *аннотация*, ассоциируется с вызывающим объектом. В противном случае возвращает значение `false`
`boolean isCompatibleWith(String номерВерсии) throws NumberFormatException`	Возвращает значение `true`, если *номерВерсии* меньше номера версии вызывающего пакета или равно ему
`boolean isSealed()`	Возвращает значение `true`, если вызывающий пакет "запечатан" (sealed). В противном случае возвращает значение `false`
`boolean isSealed(URL url)`	Возвращает значение `true`, если вызывающий пакет "запечатан" (sealed) относительно *url*. В противном случае возвращает значение `false`
`String toString()`	Возвращает строковый эквивалент вызывающего пакета

Класс RuntimePermission

Класс `RuntimePermission` относится к механизму безопасности Java и подробно здесь не рассматривается.

Класс Throwable

Класс `Throwable` поддерживает систему обработки исключений Java и представляет собой класс, от которого происходят все классы исключений. Этот класс рассматривался в главе 10.

Класс SecurityManager

Это — класс, от которого можно наследовать подклассы для создания диспетчера безопасности. В общем случае, как правило, нет необходимости реализовывать собственный диспетчер безопасности. Если вы все же решите это сделать, проконсультируйтесь с документацией, поставляемой с системой разработки Java.

Класс StackTraceElement

Класс StackTraceElement описывает единственный *стековый фрейм*, который представляет собой индивидуальный элемент трассировки стека при возникновении исключения. Каждый стековый фрейм представляет *точку выполнения*, которая включает имя класса, имя метода, имя файла и номер строки исходного кода. Массив элементов класса StackTraceElement возвращается методом getStackTrace() класса Throwable.

Класс StackTraceElement имеет один конструктор.

```
StackTraceElement(String имяКласса, String имяМетода, String имяФайла,
int строка)
```

Здесь имя класса указано в параметре *имяКласса*, имя метода — в параметре *имяМетода*, имя файла — в параметре *имяФайла*, а номер строки передается в параметре *строка*. Если нет допустимого номера строки, используйте отрицательное значение *строка*. Более того, значение -2 для параметра *строка* означает, что этот фрейм ссылается на машинно-зависимый (native) метод.

Методы, поддерживаемые классом StackTraceElement, перечислены в табл. 16.25. Они предоставляют программе доступ к ее трассировке стека.

Таблица 16.25. Методы, определенные в классе StackTraceElement

Метод	Описание
boolean equals(Object *объект*)	Возвращает значение true, если вызывающий объект класса StackTraceElement тот же, что передан в параметре *объект*. В противном случае возвращает значение false
String getClassName()	Возвращает имя класса, в котором точка выполнения описана вызывающим объектом класса StackTraceElement
String getFileName()	Возвращает имя файла, в котором исходный код точки выполнения описан хранимым вызывающим объектом класса StackTraceElement
int getLineNumber()	Возвращает номер строки исходного кода, в котором точка выполнения описана вызывающим объектом класса StackTraceElement. В некоторых ситуациях номер строки не будет доступен, тогда возвращается отрицательное значение
String getMethodName()	Возвращает имя метода в, в котором точка выполнения описана вызывающим объектом класса StackTraceElement
int hashCode()	Возвращает хеш-код вызывающего объекта класса StackTraceElement
boolean isNativeMethod()	Возвращает значение true, если точка выполнения описана вызывающим объектом класса StackTraceElement, находящимся в машинно-зависимом методе. В противном случае возвращает значение false
String toString()	Возвращает строковый эквивалент вызывающей последовательности

Класс Enum

Как было описано в главе 12, *перечисление* (enumeration) — это список именованных констант. (Вспомните, что перечисление создается ключевым словом enum.) Все перечисления автоматически наследуются от класса Enum. Класс Enum — это обобщенный класс, объявленный следующим образом.

```
class Enum<E extends Enum<E>>
```

Здесь *E* обозначает перечислимый тип. Класс Enum не имеет открытых конструкторов.

В классе Enum определено несколько методов, доступных для использования всеми перечислителями. Они описаны в табл. 16.26.

Таблица 16.26. Методы, определенные в классе Enum

Метод	Описание
`protected final Object clone() throws CloneNotSupportedException`	Вызов этого метода инициирует передачу исключения CloneNotSupportedException. Это предотвращает клонирование перечислений
`final int compareTo(E e)`	Сравнивает порядковое значение двух констант одного перечисления. Возвращает отрицательное значение, если вызывающая константа имеет порядковое значение меньше *e*, нуль — если порядковое значение совпадает с *e* и больше нуля — если значение больше *e*
`final boolean equals(Object объект)`	Возвращает значение true, если *объект* 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 вызывающего объекта
`final String name()`	Возвращает неизменяемое имя вызывающей константы
`final int ordinal()`	Возвращает значение, которое указывает позицию перечислимой константы в списке констант
`String toString()`	Возвращает имя вызывающей константы. Это имя может отличаться от того, которое использовалось при объявлении перечисления
`static <T extends Enum<T>> T valueOf(Class<T> e-тип, String имя)`	Возвращает константу, ассоциированную с *имя*, в типе перечисления, заданном *e-тип*

Класс ClassValue

Класс ClassValue, добавленный в комплекте JDK 7, применяется для связи значения с типом. Это обобщенный тип, определенный так.

```
Class ClassValue<T>
```

Он предназначен для узкоспециализированного использования, а не для обычного программирования.

Интерфейс CharSequence

Интерфейс CharSequence определяет методы, которые предоставляют доступ только для чтения к последовательности символов. Методы описаны в табл. 16.27. Этот интерфейс реализован в классах String, StringBuffer, StringBuilder и др.

Таблица 16.27. Методы, определенные в интерфейсе CharSequence

Метод	Описание
char charAt(int *индекс*)	Возвращает символ, находящийся в позиции, указанной индексом *индекс*
int length()	Возвращает количество символов в вызывающей последовательности
CharSequence subSequence(int *начИндекс*, int *конИндекс*)	Возвращает подмножество вызывающей последовательности, начиная с *начИндекс* и заканчивая *конИндекс*-1
String toString()	Возвращает строковый эквивалент вызывающей последовательности

Интерфейс Comparable

Объекты классов, реализующих интерфейс Comparable, могут быть упорядочены. Другими словами, классы, реализующие интерфейс Comparable, содержат объекты, которые можно сравнивать некоторым осмысленным образом. Интерфейс Comparable — это обобщенный интерфейс, объявленный следующим образом.

```
interface Comparable<T>
```

Здесь *T* представляет собой тип сравниваемых объектов.

Интерфейс Comparable объявляет один метод, который используется для определения того, что в языке Java называется *натуральным порядком* экземпляров класса. Сигнатура метода показана ниже.

```
int compareTo(T объект)
```

Этот метод сравнивает вызывающий объект с указанным параметром *объект*. Возвращает значение 0, если значения эквивалентны. Отрицательное значение возвращается, если вызывающий объект имеет меньшее значение. В противном случае возвращается положительное значение.

Этот интерфейс реализован в нескольких классах, уже рассмотренных ранее. В частности, классы Byte, Character, Double, Float, Long, Short, String и Integer определяют метод compareTo(). В дополнение, как показано в следующей главе, объекты, которые реализуют этот интерфейс, могут быть использованы в разных коллекциях.

Интерфейс Appendable

Объекты классов, реализующих интерфейс Appendable, позволяют добавлять к своим объектам символы или символьные последовательности. Интерфейс Appendable определяет следующие три метода.

```
Appendable append(char символ) throws IOException
Appendable append(CharSequence символы) throws IOException
Appendable append(CharSequence символы, int начало, int конец) throws
IOException
```

В первой форме символ *символ* добавляется к вызывающему объекту. Во второй форме к вызывающему объекту добавляется последовательность символов *символы*. Третья форма позволяет указать фрагмент (с символа, заданного параметром *начало*, до *конец*-1) последовательности *символы*. Во всех случаях возвращается ссылка на вызывающий объект.

Интерфейс Iterable

Интерфейс `Iterable` должен быть реализован всеми классами, объекты которых будут использованы в версии "for-each" цикла `for`. Другими словами, для того, чтобы объект использовался внутри цикла "for-each", его класс должен реализовывать интерфейс цикла `Iterable`. Интерфейс `Iterable` — это обобщенный интерфейс, имеющий следующее объявление.

```
interface Iterable<T>
```

Здесь *T* представляет собой тип объектов, по которым выполняется итерация. Он определяет один метод, `iterator()`, показанный ниже.

```
Iterator<T> iterator()
```

Метод возвращает итератор, связанный с элементами, содержащимися в вызывающем объекте.

На заметку! Итераторы рассматриваются в главе 17.

Интерфейс Readable

Интерфейс `Readable` указывает, что объект может быть использован в качестве источника символов. В этом интерфейсе определен один метод — `read()`.

```
int read(CharBuffer буфер) throws IOException
```

Этот метод читает символы в *буфер*. Возвращает количество прочитанных символов или `-1` — в случае достижения символа конца файла (EOF).

Интерфейс AutoCloseable

Интерфейс `AutoCloseable`, добавленный в JDK 7, обеспечивает поддержку для нового оператора `try`-*с-ресурсами*, реализующего то, что иногда называется *автоматическим управлением ресурсами* (Automatic Resource Management — ARM). Оператор `try`-*с-ресурсами* автоматизирует процесс освобождения ресурса (такого, как поток), когда он больше не нужен. (Более подробная информация по этой теме приведена в главе 13.) Только объекты классов, реализующих интерфейс `AutoCloseable`, могут применяться с оператором `try`-*с-ресурсами*. Интерфейс `AutoCloseable` определяет только метод `close()`, приведенный ниже.

```
void close() throws Exception
```

Этот метод закрывает вызывающий объект, освобождая любые ресурсы, которые он мог занимать. Он автоматически вызывается в конце оператора `try`-

с-ресурсами, избавляя, таким образом, от необходимости явно вызывать метод `close()`. Интерфейс `AutoCloseable` реализуется несколькими классами, включая все классы ввода-вывода, открывающие поток, который может быть закрыт.

Интерфейс `Thread.UncaughtExceptionHandler`

Статический интерфейс `Thread.UncaughtExceptionHandler` реализуется классами, которые хотят обрабатывать необработанные исключения. Его реализует класс `ThreadGroup`. В этом интерфейсе объявлен только один приведенный ниже метод.

```
void uncaughtException(Thread поток, Throwable исключение)
```

Здесь *поток* — это ссылка на поток, который создал исключение, а *исключение* — это ссылка на исключение.

Вложенные пакеты `java.lang`

В языке Java определено несколько пакетов, вложенных в пакет `java.lang`.

- `java.lang.annotation`
- `java.lang.instrument`
- `java.lang.invoke`
- `java.lang.management`
- `java.lang.ref`
- `java.lang.reflect`

Каждый из них кратко описан ниже.

Пакет `java.lang.annotation`

Средства аннотирования языка Java поддерживаются с помощью пакета `java.lang.annotation`. В этом пакете определен интерфейс `Annotation`, а также перечисления `ElementType` и `RetentionPolicy`. (Интерфейс `Annotation` описан в главе 12.)

Пакет `java.lang.instrument`

Этот пакет определяет средства, которые могут быть использованы для добавления инструментария для разных аспектов выполнения программ. Он определяет интерфейсы `Instrumentation` и `ClassFileTransformer`, а также класс `ClassDefinition`.

Пакет `java.lang.invoke`

Добавленный в комплект JDK 7, пакет `java.lang.invoke` поддерживает динамические языки. Он содержит такие классы, как `CallSite`, `MethodHandle` и `MethodType`.

Пакет java.lang.management

Этот пакет предоставляет поддержку управления виртуальной машиной Java и исполняющим окружением. Используя средства пакета java.lang.management, вы можете просматривать различные аспекты выполнения программы и управлять ими.

Пакет java.lang.ref

Ранее уже упоминалось, что средства сбора "мусора" в языке Java автоматически определяют, когда не остается ссылок на объект. Затем предполагается, что этот объект более не нужен и занятую им память можно утилизировать. Классы пакета java.lang.ref предлагают более тонкие возможности управления процессом сбора "мусора".

Пакет java.lang.reflect

Рефлексия (reflection) – это свойство программы анализировать код во время выполнения. Пакет java.lang.reflect позволяет получать информацию о полях, конструкторах, методах и модификаторах класса. Помимо других причин, эта информация может понадобиться для создания программных инструментов, которые позволяют работать с компонентами Java Beans. Инструмент использует рефлексию для динамического определения характеристик компонента. Рефлексия была представлена в главе 12 и также рассматривается в главе 28.

Пакет java.lang.reflect определяет несколько классов, включая классы Method, Field и Constructor. Также этот пакет содержит несколько интерфейсов, в числе которых AnnotatedElement, Member и Type. В дополнение пакет java.lang.reflect включает класс Array, позволяющий динамически создавать массивы и оперировать ими.

ГЛАВА 17

Пакет java.util: инфраструктура Collections Framework

Настоящая глава посвящена классам и интерфейсам, определенным в пакете `java.util`. Этот важный пакет содержит большой ассортимент классов и интерфейсов, поддерживающих широкий диапазон функциональных возможностей. Например, пакет `java.util` включает классы, создающие псевдослучайные числа, управляющие датами и временем, просмотром событий, манипулирующие наборами битов, разбирающие строки и управляющие форматированными данными. Пакет `java.util` также включает одну из наиболее мощных подсистем Java – коллекции. Инфраструктура коллекций – это сложная иерархия интерфейсов и классов, предоставляющих изящную технологию управления группами объектов. Она заслуживает пристального внимания всех программистов.

Поскольку пакет `java.util` содержит широкий диапазон функциональных возможностей, он достаточно объемный. Ниже приведен список основных его классов.

AbstractCollection	EventObject	PropertyResourceBundle
AbstractList	FormattableFlags	Random
AbstractMap	Formatter	ResourceBundle
AbstractQueue	GregorianCalendar	Scanner
AbstractSequentialList	HashMap	ServiceLoader
AbstractSet	HashSet	SimpleTimeZone
ArrayDeque	Hashtable	Stack
ArrayList	IdentityHashMap	StringTokenizer
Arrays	LinkedHashMap	Timer
BitSet	LinkedHashSet	TimerTask
Calendar	LinkedList	TimeZone
Collections	ListResourceBundle	TreeMap
Currency	Locale	TreeSet
Date	Objects (Добавлено в JDK 7)	UUID
Dictionary	Observable	Vector
EnumMap	PriorityQueue	WeakHashMap
EnumSet	Properties	
EventListenerProxy	PropertyPermission	

В пакете `java.util` определены следующие интерфейсы.

Collection	List	Queue
Comparator	ListIterator	RandomAccess
Deque	Map	Set

Enumeration	Map.Entry	SortedMap
EventListener	NavigableMap	SortedSet
Formattable	NavigableSet	
Iterator	Observer	

Из-за большого размера пакета java.util его описание разделено на две главы. Эта глава посвящена средствам инфраструктуры коллекций Collections Framework пакета java.util. В главе 18 рассматриваются остальные классы и интерфейсы этого пакета.

Обзор коллекций

Инфраструктура коллекций Java Collections Framework стандартизирует способы управления группами объектов для ваших программ. Коллекции не были частью исходной версии языка Java, но были добавлены в комплекте J2SE 1.2. До появления инфраструктуры коллекций для хранения групп объектов и манипулирования ими язык Java предлагал такие специальные классы, как Dictionary, Vector, Stack и Properties. Хотя эти классы были достаточно удобны, им недоставало централизованной, универсальной, идеи. Так, например, способ применения класса Vector отличался от способа использования класса Properties. Кроме того, этот ранний, специализированный подход не был спроектирован в расчете на дальнейшее расширение и адаптацию. Коллекции стали решением этой и ряда других проблем.

Инфраструктура коллекций была разработана для достижения нескольких целей. Во-первых, она должна была обеспечить высокую производительность. Реализация основных коллекций (динамических массивов, связных списков, деревьев и хеш-таблиц) отличается высокой эффективностью. Очень редко вам понадобится (если вообще понадобится) программировать один из таких "механизмов данных" вручную. Во-вторых, эта система должна была позволить разным типам коллекций работать в единой манере и с высокой степенью взаимодействия. В-третьих, расширение и/или адаптация коллекций должны были быть просты. И наконец, вся инфраструктура коллекций построена на едином наборе стандартных интерфейсов. Некоторые стандартные реализации (такие, как классы LinkedList, HashSet и TreeSet) этих интерфейсов вы можете использовать "как есть". Вы также можете реализовать свои собственные коллекции, если хотите. Для вашего удобства предусмотрены различные реализации специального назначения, а также частичные реализации, которые облегчают создание ваших собственных коллекций. Наконец, в систему коллекций были добавлены механизмы интеграции стандартных массивов.

Алгоритмы — это другая важная часть инфраструктуры коллекций. Алгоритмы оперируют коллекциями и определены как статические методы класса Collections. Таким образом, они доступны всем коллекциям. Каждый класс коллекции не нуждается в реализации собственной версии. Алгоритмы представляют стандартные способы манипулирования коллекциями.

Другая сущность, тесно связанная с системой коллекций, — это интерфейс итератора Iterator. *Итератор* (iterator) предоставляет общий, стандартизированный способ доступа к элементам коллекций по одному. То есть итератор предлагает способ *перебора содержимого коллекций*. Поскольку каждая коллекция предоставляет итератор, элементы любого класса коллекций могут быть доступны через методы, определенные в интерфейсе Iterator. Таким образом, код, перебирающий в цикле *набор* (set), с минимальными изменениями можно применить, например, к *списку* (list).

В дополнение к коллекциям в инфраструктуре определено также несколько интерфейсов и классов карт. *Карта* (map) хранит пары "ключ-значение". Хотя карты являются частью системы коллекций, они, строго говоря, коллекциями не явля-

ются. Тем не менее вы можете получить доступ к картам в виде коллекций. Такое представление содержит элементы карты, помещенные в коллекцию. Таким образом, вы, при желании, можете обрабатывать содержимое карты как коллекцию.

Механизм коллекций был модифицирован для некоторых классов, изначально определенных в пакете `java.util` таким образом, что они также могут быть интегрированы в новую систему. Важно понимать: несмотря на то что добавление коллекций изменило архитектуру многих оригинальных служебных классов, это не отменило ни одного из них. Коллекции просто предлагают лучший способ выполнения некоторых задач.

На заметку! Если вы знакомы с языком C++, то, возможно, обнаружите, что вам поможет сходство между технологией коллекций Java и идеологией *стандартной библиотеки шаблонов* (Standard Template Library — STL), определенной в C++. То, что в C++ называется контейнером, в Java именуется коллекцией. Однако есть существенные отличия между системой коллекций и STL. Поэтому важно не делать поспешных выводов.

Комплект JDK 5 изменил инфраструктуру Collections Framework

Когда вышел комплект JDK 5, в инфраструктуре Collections Framework произошло несколько фундаментальных изменений, значительно повысивших ее мощь и упростивших ее применение. К этим изменениям относится добавление обобщений, автоматическая упаковка и распаковка, а также стиль "for-each" цикла `for`. Хотя комплект JDK 7 является следующим основным выпуском языка Java после JDK 5, эффект от средств комплекта JDK 5 был настолько глубок, что они все еще заслуживают особого внимания. Основная причина в том, что все еще существует и используется большое количество кода, написанного до появления JDK 5. Понимание роли и причин этих изменений очень важно, если вы будете обслуживать или модифицировать устаревший код.

Обобщенные определения фундаментально изменили инфраструктуру коллекций

Добавление обобщенных определений — существенное изменение в инфраструктуре коллекций, поскольку для этого она полностью была перепроектирована. Все коллекции теперь обобщенные, и многие методы, оперирующие коллекциями, также принимают обобщенные параметры. Простое добавление этого свойства коснулось всех частей инфраструктуры коллекций.

Обобщения — это то, чего не хватало коллекциям. Это — безопасность типов. Раньше все коллекции хранили ссылки на класс `Object`, а это означало, что любая коллекция могла хранить объекты любого типа. Таким образом, можно было непреднамеренно сохранить несовместимые типы в одной коллекции. Это могло привести к ошибкам несовместимости типов во время выполнения. С обобщенными определениями можно явно указать тип сохраняемых данных, и таких ошибок времени выполнения можно избежать.

Несмотря на то что добавление обобщенных определений изменило объявления большинства классов и интерфейсов, а также некоторых их методов, в общем, инфраструктура коллекций по-прежнему работает так же, как и ранее. Но если вы знакомы со старыми версиями инфраструктуры коллекций, новый синтаксис мо-

жет показаться несколько пугающим. Не беспокойтесь, со временем обобщенный синтаксис станет привычным.

Еще один момент: чтобы получить выгоду от этого нововведения в коллекциях, старый код должен быть переписан. Это также важно, поскольку в противном случае старый код при компиляции современным компилятором Java будет создавать предупреждения. Чтобы исключить эти сообщения, вам придется добавить информацию о типе везде, где у вас встречается код, работающий с коллекциями.

Средства автоматической упаковки используют элементарные типы

Сохранение элементарных типов в коллекциях облегчает автоматическая упаковка. Как вы увидите, коллекции могут сохранять только ссылки, но не элементарные значения. Раньше, если вы хотели сохранять в коллекции значение элементарного типа вроде `int`, то должны были вручную упаковать его в объект-оболочку типа. Когда значение извлекалось, нужно было его вручную распаковать (применяя явное приведение) в корректный элементарный тип. Благодаря автоматической упаковке-распаковке, язык Java теперь может делать это автоматически, когда необходимо сохранять или извлекать элементарные типы. Нет необходимости вручную делать эти операции.

Стиль цикла "for-each"

Все классы коллекций модифицированы таким образом, что реализуют интерфейс `Iterable`. Это значит, что можно перебрать содержимое коллекции, используя стиль "for-each" цикла `for`. Раньше для перебора коллекции необходимо было использовать итератор (описанный далее в настоящей главе), программно конструируя цикл. Хотя итераторы все еще применяются для некоторых целей, во многих случаях циклы на основе итераторов могут быть заменены циклами `for`.

Интерфейсы коллекций

Инфраструктура коллекций определяет несколько интерфейсов. В этом разделе представлен обзор каждого из них. Начинать с интерфейсов коллекций необходимо потому, что они определяют фундаментальную природу классов коллекций. Взятые по отдельности, конкретные классы просто представляют различные реализации стандартных интерфейсов. Интерфейсы, которые поддерживают коллекции, перечислены в табл. 17.1.

Таблица 17.1. Интерфейсы, поддерживаемые коллекциями

Интерфейс	Описание
`Collection`	Позволяет работать с группами объектов. Это — вершина иерархии коллекций
`Deque`	Расширяет интерфейс `Queue` для обработки двунаправленных очередей
`List`	Расширяет интерфейс `Collection` для управления последовательностями (списками объектов)
`NavigableSet`	Расширяет интерфейс `SortedSet` для обработки извлечения элементов на основе поисков по ближайшему соответствию

Окончание табл. 17.1

Интерфейс	Описание
`Queue`	Расширяет интерфейс `Collection` для управления специальными типами списков, в которых элементы удаляются только с начала
`Set`	Расширяет интерфейс `Collection` для управления наборами, которые должны содержать уникальные элементы
`SortedSet`	Расширяет интерфейс `Set` для управления сортированными наборами

В дополнение к этим интерфейсам, коллекции также используют интерфейсы `Comparator`, `RandomAccess`, `Iterator` и `ListIterator`, которые детально рассматриваются далее в этой главе. Короче говоря, интерфейс `Comparator` определяет два сравниваемых объекта; интерфейсы `Iterator` и `ListIterator` перечисляют объекты в коллекции. Реализуя интерфейс `RandomAccess`, список поддерживает эффективный произвольный доступ к своим элементам.

Чтобы обеспечить максимальную гибкость в применении, интерфейсы коллекций позволяют некоторым методам быть необязательными. Необязательные методы позволяют модифицировать содержимое коллекций. Коллекции, которые поддерживают эти методы, называются *модифицируемыми*. Коллекции, которые не позволяют изменять собственное содержимое, называются *немодифицируемыми*. Если предпринимается попытка вызвать один из этих методов для немодифицируемой коллекции, передается исключение `UnsupportedOperationException`. Все встроенные коллекции модифицируемы.

Следующий раздел посвящен интерфейсам коллекций.

Интерфейс `Collection`

Этот интерфейс является фундаментом, на котором построена вся инфраструктура коллекций, поскольку он должен быть реализован всеми классами коллекций. Интерфейс `Collection` — это обобщенный интерфейс, имеющий следующее объявление.

```
interface Collection<E>
```

Здесь *E* указывает тип объектов, которые будет содержать коллекция. Интерфейс `Collection` расширяет интерфейс `Iterable`. Это значит, что все коллекции можно перебирать циклами вида "for-each". (Вспомните, что только те классы, которые реализуют интерфейс `Iterable`, позволяют перебирать их элементы циклом `for`.)

Интерфейс `Collection` определяет основные методы, которые будут иметь все коллекции. Эти методы представлены в табл. 17.2.

Таблица 17.2. Методы, определенные в интерфейсе `Collection`

Метод	Описание
`boolean add(E` *объект*`)`	Добавляет *объект* к вызывающей коллекции. Возвращает значение `true`, если *объект* был добавлен к коллекции
`boolean addAll(Collection<? extends E>` *c*`)`	Добавляет все элементы *c* к вызывающей коллекции. Возвращает значение `true`, если коллекция изменена (т.е. все элементы добавлены). В противном случае возвращает значение `false`
`void clear()`	Удаляет все элементы вызывающей коллекции

Окончание табл. 17.2

Метод	Описание
`boolean contains(Object` *объект*`)`	Возвращает значение `true`, если *объект* является элементом вызывающей коллекции. В противном случае возвращает значение `false`
`boolean containsAll(Collection<?>` *c*`)`	Возвращает значение `true`, если вызывающая коллекция содержит все элементы *c*. В противном случае возвращает значение `false`
`boolean equals(Object` *объект*`)`	Возвращает значение `true`, если вызывающая коллекция и *объект* эквивалентны. В противном случае возвращает значение `false`
`int hashCode()`	Возвращает хеш-код вызывающей коллекции
`boolean isEmpty()`	Возвращает значение `true`, если вызывающая коллекция пуста. В противном случае возвращает значение `false`
`Iterator<E> iterator()`	Возвращает итератор для вызывающей коллекции
`boolean remove(Object` *объект*`)`	Удаляет один экземпляр *объект* из вызывающей коллекции. Возвращает значение `true`, если элемент удален. В противном случае возвращает значение `false`
`boolean removeAll(Collection<?>` *c*`)`	Удаляет все элементы *c* из вызывающей коллекции. Возвращает значение `true`, если в результате коллекция изменяется (т.е. элементы удалены). В противном случае возвращает значение `false`
`boolean retainAll(Collection<?>` *c*`)`	Удаляет все элементы, кроме входящих в *c* из вызывающей коллекции. Возвращает значение `true`, если в результате коллекция изменяется (т.е. элементы удалены). В противном случае возвращает значение `false`
`int size()`	Возвращает количество элементов, содержащихся в коллекции
`Object[] toArray()`	Возвращает массив, содержащий все элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции
`<T> T[] toArray(T array[])`	Возвращает массив, содержащий элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции. Если размер массива `array` равен количеству элементов, он возвращается. Если размер массива `array` меньше количества элементов, создается и возвращается новый массив нужного размера. Если размер массива `array` больше количества элементов, то элементы, следующие за последним из коллекции, устанавливаются равными `null`. Если любой элемент коллекции имеет тип, не являющийся подтипом массива `array`, передается исключение `ArrayStoreException`

Поскольку все коллекции реализуют интерфейс `Collection`, знакомство с его методами необходимо для четкого понимания инфраструктуры коллекций. Некоторые из этих методов могут передавать исключение `UnsupportedOperationException`. Как уже объяснялось, это происходит, если коллекция не может быть модифицирована. Исключение `ClassCastException` передается, когда объекты несовместимы

между собой, как, например, в случае, когда предпринимается попытка добавить несовместимый объект в коллекцию. Исключение `NullPointerException` передается при попытке вставить значение `null` в коллекцию, не допускающую пустых элементов. Исключение `IllegalArgumentException` передается при использовании неправильного аргумента.

Исключение `IllegalStateException` передается при попытке вставить новый элемент в заполненную коллекцию фиксированной длины.

Объекты добавляются в коллекции методом `add()`. Отметим, что метод `add()` принимает аргумент типа *E*. Следовательно, добавляемые в коллекцию объекты должны быть совместимыми с ожидаемым типом данных коллекции. Вы можете добавить все содержимое одной коллекции к другой вызовом метода `addAll()`.

Вы можете удалить объект, используя метод `remove()`. Чтобы удалить группу объектов, вызовите метод `removeAll()`. Можно также удалить все объекты, кроме указанных, применив метод `retainAll()`. Для очистки коллекции потребуется вызвать метод `clear()`.

Вы можете определить, содержит ли коллекция определенный объект, вызвав метод `contains()`. Чтобы определить, содержит ли одна коллекция все члены другой, вызовите метод `containsAll()`. Определить, пуста ли коллекция, можно с помощью метода `isEmpty()`. Количество элементов, содержащихся в данный момент в коллекции, возвращает метод `size()`.

Методы `toArray()` возвращают массив, который содержит элементы, хранящиеся в коллекции. Первый из них возвращает массив класса `Object`. Второй — массив элементов того типа, что и массив, указанный в параметре. Обычно второй метод более предпочтителен, поскольку он возвращает массив элементов нужного типа. Эти методы важнее, чем может показаться на первый взгляд. Часто обработка содержимого коллекции с применением синтаксиса массивов выгодна. Имея простой способ превращения коллекций в массивы, вы имеете доступ к преимуществам обоих представлений.

Две коллекции можно сравнить на эквивалентность, вызвав метод `equals()`. Точный смысл "эквивалентности" может зависеть от конкретной коллекции. Например, вы можете реализовать метод `equals()` так, чтобы он сравнивал значения элементов, хранимых в коллекции. В качестве альтернативы методу `equals()` может сравнивать ссылки на эти элементы.

Еще один очень важный метод — это `iterator()`, который возвращает итератор коллекции. Итераторы очень часто используются при работе с коллекциями.

Интерфейс `List`

Этот интерфейс расширяет интерфейс `Collection` и определяет такое поведение коллекций, которое сохраняет последовательность элементов. Элементы могут быть вставлены или извлечены по их позиции в списке с помощью индекса, начинающегося с нуля. Список может содержать повторяющиеся элементы. Интерфейс `List` — это обобщенный интерфейс, объявленный следующим образом.

```
interface List<E>
```

Здесь *E* указывает тип объектов, которые должен содержать список.

В дополнение к объявленным в интерфейсе `Collection` методам, интерфейс `List` определяет некоторые собственные методы, которые приведены в табл. 17.3. Еще раз отметим, что некоторые из этих методов передают исключение `UnsupportedOperationException`, если коллекция не может быть модифицирована, а исключение `ClassCastException` передается при несовместимости одного объекта с другим, как, например, когда осуществляется попытка добавить в спи-

сок элемент несовместимого типа. Кроме того, некоторые методы передают исключение IndexOutOfBounsException, если используется неправильный индекс. Исключение NullPointerException передается при попытке вставить в список объект null, когда пустые элементы в данном списке не допускаются. Исключение IllegalArgumentException передается при передаче неверного аргумента.

Таблица 17.3. Методы, определенные в интерфейсе List

Метод	Описание
`void add(int` *индекс*`,` `E` *объект*`)`	Вставляет *объект* в позицию вызывающего списка, указанную в параметре *индекс*. Любые ранее вставленные элементы за указанной позицией вставки смещаются вверх. То есть никакие элементы не перезаписываются
`boolean addAll(int` *индекс*`,` `Collection<? extends E>` *c*`)`	Вставляет все элементы *c* в вызывающий список, начиная с позиции, переданной в *индекс*. Все ранее существовавшие элементы за точкой вставки смещаются вверх. То есть никакие элементы не перезаписываются. Возвращает значение `true`, если вызывающий список изменяется, и значение `false` – в противном случае
`E get(int` *индекс*`)`	Возвращает объект, сохраненный в указанной позиции вызывающего списка
`int indexOf(Object` *объект*`)`	Возвращает индекс первого экземпляра *объект* в вызывающем списке. Если *объект* не содержится в списке, возвращается значение `-1`
`int lastIndexOf(Object` *объект*`)`	Возвращает индекс последнего экземпляра *объект* в вызывающем списке. Если *объект* не содержится в списке, возвращается значение `-1`
`ListIterator<E>` `listIterator()`	Возвращает итератор, указывающий на начало списка
`ListIterator<E>` `listIterator(int` *индекс*`)`	Возвращает итератор, указывающий на заданную позицию в списке
`E remove(int` *индекс*`)`	Удаляет элемент из вызывающего списка в позиции *индекс* и возвращает удаленный элемент. Результирующий список уплотняется, т.е. элементы, следующие за удаленным, сдвигаются на одну позицию назад
`E set(int` *индекс*`,` `E` *объект*`)`	Присваивает *объект* элементу, находящемуся в списке в позиции *индекс*. Возвращает прежнее значение
`List<E> subList(int` *начало*`,` `int` *конец*`)`	Возвращает список, включающий элементы от *начало* до *конец*`-1` из вызывающего списка. Элементы из возвращаемого списка также сохраняют ссылки в вызывающем списке

К версиям методов `add()` и `addAll()`, определенным в интерфейсе `Collection`, интерфейс `List` добавляет методы `add(int, E)` и `addAll(int, Collection)`. Эти методы вставляют элементы в позицию, указанную индексом.

Также семантика методов `add(E)` и `addAll(Collection)`, определенная в интерфейсе `Collection`, изменяется в интерфейсе `List` таким образом, что они добавляют элементы в конец списка.

Чтобы получить объект, сохраненный в определенной позиции, вызовите метод `get()` с индексом объекта. Чтобы присвоить значение элементу списка, вызовите метод `set()`, указав индекс объекта, который требуется изменить. Чтобы найти индекс объекта, примените метод `indexOf()` или `lastIndexOf()`.

Вы можете получить список из данного списка, вызвав метод `subList()` и указав начальный и конечный индексы нового списка. Как вы можете представить, метод `subList()` обеспечивает значительное удобство работы со списками.

Интерфейс `Set`

Интерфейс `Set` определяет набор. Он расширяет интерфейс `Collection` и определяет поведение коллекций, не допускающих дублирования элементов. Таким образом, метод `add()` возвращает значение `false` при попытке добавить в набор дублированный элемент. Он не определяет никаких собственных дополнительных методов. Интерфейс `Set` — это обобщенный интерфейс, который объявлен следующим образом.

```
interface Set<E>
```

Здесь *E* указывает тип объектов, которые должен содержать набор.

Интерфейс `SortedSet`

Интерфейс `SortedSet` расширяет интерфейс `Set` и объявляет поведение наборов, отсортированных в порядке возрастания. Интерфейс `SortedSet` — это обобщенный интерфейс, который имеет следующее объявление.

```
interface SortedSet<E>
```

Здесь *E* указывает тип объектов, которые должен содержать набор.

В дополнение к методам, предоставляемым интерфейсом `Set`, интерфейс `SortedSet` объявляет методы, перечисленные в табл. 17.4. Некоторые из них передают исключение `NoSuchElementException`, когда никаких элементов в вызывающем наборе не содержится. Исключение `ClassCastException` передается при несовместимости объекта с элементами набора. Исключение `NullPointerException` передается при попытке использовать пустой объект, когда значение `null` в наборе недопустимо. При использовании неправильного аргумента передается исключение `IllegalArgumentException`.

Таблица 17.4. Методы, определенные в интерфейсе `SortedSet`

Метод	Описание
`Comparator<? super E> comparator()`	Возвращает компаратор отсортированного набора. Если для набора применяется естественный порядок сортировки, возвращается значение `null`
`E first()`	Возвращается первый элемент вызывающего отсортированного набора
`SortedSet<E> headSet(E конец)`	Возвращается объект интерфейса `SortedSet`, содержащий элементы из вызывающего набора, которые предшествуют *конец*. Элементы из возвращенного набора имеют также ссылки в вызывающем объекте
`E last()`	Возвращается последний элемент вызывающего отсортированного набора

Окончание табл. 17.4

Метод	Описание
`SortedSet<E> subSet(E начало, E конец)`	Возвращается объект интерфейса `SortedSet`, который включает элементы, находящиеся между *начало* и *конец*-1. Элементы из возвращенного набора имеют также ссылки в вызывающем объекте
`SortedSet<E> tailSet(E начало)`	Возвращается объект интерфейса `SortedSet`, содержащий элементы из вызывающего набора, которые следуют за *конец*. Элементы из возвращенного набора имеют также ссылки в вызывающем объекте

В интерфейсе `SortedSet` определено несколько методов, которые облегчают обработку. Чтобы получить первый объект в отсортированном наборе, вызовите метод `first()`, а чтобы последний — метод `last()`. Вы можете получить набор из отсортированного набора, вызвав метод `subSet()` и передав ему первый и последний объекты набора. Если вам нужно получить набор, который начинается с первого элемента существующего набора, используйте метод `headSet()`. Если же требуется получить набор из конца существующего набора, то вызовите метод `tailSet()`.

Интерфейс `NavigableSet`

Этот интерфейс расширяет интерфейс `SortedSet` и объявляет поведение коллекции, которая поддерживает извлечение элементов на основе ближайшего соответствия заданному значению или значениям. Интерфейс `NavigableSet` — это обобщенный интерфейс, имеющий следующее объявление.

```
interface NavigableSet<E>
```

Здесь *E* определяет тип содержащихся в наборе объектов. В дополнение к методам, унаследованным от интерфейса `SortedSet`, интерфейс `NavigableSet` включает также и те, что перечислены в табл. 17.5. Исключение `ClassCastException` передается при несовместимости с элементами набора. Исключение `NullPointerException` передается при попытке вставить пустой объект, когда набор не допускает значений `null`. При использовании неправильного аргумента передается исключение `IllegalArgumentException`.

Таблица 17.5. Методы, определенные в интерфейсе `NavigableSet`

Метод	Описание
`E ceiling(E объект)`	Ищет в наборе наименьший элемент *e*, для которого истинно *e*>=*объект*. Если такой элемент найден, он возвращается. В противном случае возвращается значение `null`
`Iterator<E> descendingIterator()`	Возвращает итератор, перемещающийся от большего к меньшему, другими словами, обратный итератор
`NavigableSet<E> descendingSet()`	Возвращает объект интерфейса `NavigableSet`, представляющий собой обратную версию вызывающего набора. Результирующий набор поддерживается вызывающим набором
`E floor(E объект)`	Ищет в наборе наибольший элемент *e*, для которого истинно *e*<=*объект*. Если такой элемент найден, он возвращается. В противном случае возвращается значение `null`

Окончание табл. 17.5

Метод	Описание
`NavigableSet<E> headSet(E` *верхнГраница*`, boolean` *включать*`)`	Возвращает объект интерфейса `NavigableSet`, включающий все элементы вызывающего набора, меньшие *верхнГраница*. Результирующий набор поддерживается вызывающим набором
`E higher(E` *объект*`)`	Ищет в наборе наибольший элемент *e*, для которого истинно *e*>*объект*. Если такой элемент найден, он возвращается. В противном случае возвращается значение `null`
`E lower(E` *объект*`)`	Ищет в наборе наименьший элемент *e*, для которого истинно *e*<*объект*. Если такой элемент найден, он возвращается. В противном случае возвращается значение `null`
`E pollFirst()`	Возвращает первый элемент, удаляя его в процессе. Поскольку набор отсортирован, это будет элемент с наименьшим значением. Возвращает значение `null` в случае пустого набора
`E pollLast()`	Возвращает последний элемент, удаляя его в процессе. Поскольку набор отсортирован, это будет элемент с наибольшим значением. Возвращает значение `null` в случае пустого набора
`NavigableSet<E> subSet(E` *нижнГраница*`, boolean` *включатьНижн*`, E` *верхнГраница*`, boolean` *включатьВерхн*`)`	Возвращает объект интерфейса `NavigableSet`, включающий все элементы вызывающего набора, которые больше *нижнГраница* и меньше *верхнГраница*. Если параметр *включатьНижн* содержит значение `true`, то элемент, равный *нижнГраница*, включается. Если параметр *включатьВерхн* содержит значение `true`, также включается элемент, равный *верхнГраница*
`NavigableSet<E> tailSet(E` *нижнГраница*`, boolean` *включать*`)`	Возвращает объект интерфейса `NavigableSet`, включающий все элементы из вызывающего набора, которые больше *нижнГраница*. Если параметр *включать* содержит значение `true`, в результат включается элемент, равный *нижнГраница*. Результирующий набор поддерживается вызывающим набором

Интерфейс `Queue`

Этот интерфейс расширяет интерфейс `Collection` и объявляет поведение очередей, которые представляют собой список по принципу "первый вошел — первый вышел". Однако существуют разные типы очередей, в которых порядок основан на некотором критерии. Интерфейс `Queue` — это обобщенный интерфейс со следующим объявлением.

```
interface Queue<E>
```

Здесь *E* указывает тип объектов, которые будут храниться в очереди. Методы, определенные в интерфейсе `Queue`, представлены в табл. 17.6.

Несколько методов передают исключение `ClsssCastException`, когда объект несовместим с элементами очереди. Исключение `NullPointerException` передается при попытке сохранения пустого объекта, когда пустые элементы в очереди не разрешены. Исключение `IllegalArgimentException` передается при использовании неверного аргумента. Исключение `IllegalStateException` передается при попытке вставки в полную очередь фиксированной длины. Исключение `NoSuchElementException` передается при попытке удалить элемент из пустой очереди.

Несмотря на свою простоту, интерфейс Queue представляет интерес с нескольких точек зрения. Во-первых, элементы могут удаляться только из начала очереди. Во-вторых, есть два метода, которыми можно получать и удалять элементы, — poll() и remove().

Таблица 17.6. Методы, определенные в интерфейсе Queue

Метод	Описание
E element()	Возвращает элемент из головы очереди. Элемент не удаляется. Если очередь пуста, передается исключение NoSuchElementException
boolean offer(E *объект*)	Пытается добавить *объект* в очередь. Возвращает значение true, если *объект* добавлен, и значение false — в противном случае
E peek()	Возвращает элемент из головы очереди. Возвращает значение null, если очередь пуста. Элемент не удаляется
E poll()	Возвращает элемент из головы очереди и удаляет его. Возвращает значение null, если очередь пуста
E remove()	Удаляет элемент из головы очереди, возвращая его. Инициирует исключение NoSuchElementException, если очередь пуста

Разница между ними в том, что метод poll() возвращает значение null, если очередь пуста, а метод remove() передает исключение. В-третьих, есть два метода, element() и peek(), которые получают элемент из головы очереди, но не удаляют его. Отличаются они тем, что при пустой очереди метод element() передает исключение, а метод peek() возвращает значение null. И наконец, отметим, что метод offer() только пытается добавить элемент в очередь. Поскольку некоторые очереди имеют фиксированную длину и могут быть заполнены, метод offer() может завершиться неудачно.

Интерфейс Dequeue

Интерфейс Dequeue расширяет интерфейс Queue и описывает поведение двунаправленной очереди, которая может функционировать как стандартная очередь "первый вошел — первый вышел" либо как стек "последний вошел — первый вышел". Интерфейс Dequeue — это обобщенный интерфейс со следующим объявлением.

```
interface Deque<E>
```

Здесь *E* определяет тип объектов, которые будет содержать двусторонняя очередь.

В дополнение к методам, унаследованным от интерфейса Queue, интерфейс Dequeue добавляет методы, перечисленные в табл. 17.7. Несколько методов передают исключение ClassCastException, когда объект несовместим с элементами в двунаправленной очереди. Исключение NullPointerException передается при попытке сохранения пустого объекта, когда пустые элементы двунаправленной очереди не допускаются. При использовании неверного аргумента передается исключение IllegalArgimentException. Исключение IllegalStateException передается при попытке вставки в полную двунаправленную очередь фиксированной длины. Исключение NoSuchElementException передается при попытке удалить элемент из пустой очереди.

Таблица 17.7. Методы, определенные в интерфейсе Dequeue

Метод	Описание
void addFirst(E *объект*)	Добавляет *объект* в голову двунаправленной очереди. Передает исключение IllegalStateException, если в очереди фиксированной длины нет места
void addLast(E *объект*)	Добавляет *объект* в хвост двунаправленной очереди. Передает исключение IllegalStateException, если в очереди фиксированной длины нет места
Iterator<E> descendingIterator()	Возвращает итератор, перемещающийся от хвоста к голове двунаправленной очереди. То есть возвращает обратный итератор
E getFirst()	Возвращает первый элемент двунаправленной очереди. Объект из очереди не удаляется. В случае пустой двунаправленной очереди передает исключение NoSuchElementException
E getLast()	Возвращает последний элемент двунаправленной очереди. Объект из очереди не удаляется. В случае пустой двунаправленной очереди передает исключение NoSuchElementException
boolean offerFirst(E *объект*)	Пытается добавить *объект* в голову двунаправленной очереди. Возвращает значение true, если *объект* добавлен, и значение false — в противном случае. Таким образом, этот метод возвращает значение false при попытке добавить *объект* в полную двунаправленную очередь фиксированной длины
boolean offerLast(E *объект*)	Пытается добавить *объект* в хвост двунаправленной очереди. Возвращает значение true, если *объект* добавлен, и значение false — в противном случае
E peekFirst()	Возвращает элемент, находящийся в голове двунаправленной очереди. Возвращает значение null, если очередь пуста. Объект из очереди не удаляется
E peekLast()	Возвращает элемент, находящийся в хвосте двунаправленной очереди. Возвращает значение null, если очередь пуста. Объект из очереди не удаляется
E pollFirst()	Возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возвращает значение null, если очередь пуста
E pollLast()	Возвращает элемент, находящийся в хвосте двунаправленной очереди, одновременно удаляя его из очереди. Возвращает значение null, если очередь пуста
E pop()	Возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Передает исключение NoSuchElementException, если очередь пуста
void push(E *объект*)	Добавляет элемент в голову двунаправленной очереди. Если в очереди фиксированной длины нет места, передает исключение IllegalStateException
E removeFirst()	Возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Передает исключение NoSuchElementException, если очередь пуста

Обратите внимание на то, что интерфейс Deque включает методы push() и pop(), которые позволяют интерфейсу Deque функционировать в качестве стека. Кроме того, следует обратить внимание на метод descendingIterator(). Он возвращает итератор, который возвращает элементы в обратном порядке. Другими сло-

вами, итератор, перемещающийся от конца коллекции к ее началу. Реализация интерфейса Deque может быть ограниченной по емкости, т.е. в него может быть добавлено ограниченное количество элементов. В этом случае попытка добавления элемента может вызвать исключение. Интерфейс Deque позволяет обрабатывать такие сбои двумя способами. Во-первых, методы вроде addFirst() и addLast() передают исключение IllegalStateException, если двунаправленная очередь имеет ограниченную емкость. Во-вторых, такие методы, как offerFirst() и offerLast(), возвращают значение false, когда 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 представляют полную реализацию и могут применяться "как есть". Другие же являются абстрактными, представляя только шаблонные реализации, которые используются в качестве начальных точек при создании конкретных коллекций. Как правило, классы коллекций не синхронизированы, но, как будет показано далее в настоящей главе, при необходимости можно получить их синхронизированные версии.

Стандартные классы коллекций перечислены в табл. 17.8.

Таблица 17.8. Стандартные классы коллекций

Класс	Описание
AbstractCollection	Реализует большую часть интерфейса Collection
AbstractList	Расширяет класс AbstractCollection и реализует большую часть интерфейса List
AbstractQueue	Расширяет класс AbstractCollection и реализует часть интерфейса Queue
AbstractSequentalList	Расширяет класс AbstractList для использования в коллекциях, использующих последовательности вместо случайного доступа к элементам
LinkedList	Реализует связный список, расширяя класс AbstractSequentalList
ArrayList	Реализует динамический массив, расширяя класс AbstractList
ArrayDeque	Реализует динамическую двухстороннюю очередь, расширяя класс AbstractCollection и реализуя интерфейс Deque
AbstractSet	Расширяет класс AbstractCollection и реализует большую часть интерфейса Set
EnumSet	Расширяет класс AbstractSet для использования с элементами типа enum
HashSet	Расширяет класс AbstractSet для использования с хеш-таблицами
LinkedHashSet	Расширяет класс HashSet, разрешая итерации порядковой вставки
PriorityQueue	Расширяет класс AbstractQueue для поддержки очередей, основанных на приоритетах
TreeSet	Реализует набор, хранимый в дереве. Расширяет класс AbstractSet

В следующих разделах рассматриваются конкретные классы коллекций и иллюстрируется их применение.

На заметку! В дополнение к классам коллекций, некоторые унаследованные от прежних версий классы, такие как `Vector`, `Stack` и `HashTable`, были перепроектированы для поддержки коллекций. Они также рассматриваются далее в этой главе.

Класс `ArrayList`

Класс `ArrayList` расширяет класс `AbstractList` и реализует интерфейс `List`. Класс `ArrayList` — это обобщенный класс со следующим объявлением.

```
class ArrayList<E>
```

Здесь параметр *E* указывает тип сохраняемых объектов.

Класс `ArrayList` поддерживает динамические массивы, которые могут расти по мере необходимости. Стандартные массивы Java имеют фиксированную длину. После того как массив создан, он не может увеличиваться или уменьшаться, а это значит, что следует заранее знать, сколько элементов нужно в нем хранить. Но иногда вы не можете точно знать до момента выполнения программы, насколько большой массив понадобится. Чтобы справиться с этой ситуацией, в инфраструктуре коллекций определен класс `ArrayList`. По сути, класс `ArrayList` — это массив объектных ссылок переменной длины. То есть объект класса `ArrayList` может динамически увеличиваться или уменьшаться в размере. Массивы-списки (ArrayList) создаются с некоторым начальным размером. Когда этот первоначальный размер становится недостаточным, коллекция автоматически увеличивается. Когда объекты удаляются, коллекция может уменьшаться.

На заметку! Динамические массивы поддерживаются также унаследованным классом `Vector`, который будет описан далее в главе.

Класс `ArrayList` имеет следующие конструкторы.

```
ArrayList()
ArrayList(Collection <? extends E> c)
ArrayList(int емкость)
```

Первый конструктор создает пустой массив-список. Второй создает массив-список, который инициализируется элементами коллекции *c*. Третий конструктор создает массив-список, который имеет начальную емкость *емкость*. Емкость — это размер лежащего в основе массива, используемого для хранения элементов. Емкость растет автоматически по мере добавления элементов в массив-список.

В следующей программе демонстрируется простое применение класса `ArrayList`. Создается массив-список для объектов класса `String`, затем в него добавляется несколько строк. (Вспомните, что строки в кавычках транслируются в объекты класса `String`.) После этого отображается список. Некоторые элементы удаляются, и список отображается снова.

```
/// Демонстрация использования ArrayList.
import java.util.*;

class ArrayListDemo {
    public static void main (String args[]) {
        // Создать массив-список.
        ArrayList<String> al = new ArrayList<String>();

        System.out.println("Начальный размер al: " + al.size());
```

```
        // Добавить элементы в массив-список.
        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 вставки: " +
                           al.size());

        // отобразить массив-список.
        System.out.println("Содержимое al: " + al);

        // Удалить элементы из массива-списка.
        al.remove("F");
        al.remove(2);

        System.out.println("Размер al после удалений: " + al.size());

        System.out.println("Содержимое al: " + al);
    }
}
```

Вывод программы показан ниже.

```
Начальный размер al: 0
Размер al после вставки: 7
Содержимое al: [C, A2, A, E, B, D, F]
Размер al после удалений: 5
Содержимое al: [C, A2, E, B, D]
```

Обратите внимание на то, что массив-список `al` изначально пуст и растет по мере добавления в него элементов. Когда элементы удаляются, его размер уменьшается.

В предыдущем примере содержимое коллекции отображается с использованием преобразования типов по умолчанию, предоставляемого методом `toString()`, который унаследован от класса `AbstractCollection`. Несмотря на то что это удобно при написании коротких примеров программ, вы редко будете пользоваться этим методом для отображения содержимого реальных коллекций. Обычно вы будете предусматривать свои собственные процедуры вывода. Но для нескольких следующих примеров вывод по умолчанию, выполняемый методом `toString()`, вполне подойдет.

Невзирая на то что емкость объектов класса `ArrayList` растет автоматически по мере сохранения в них объектов, вы также можете увеличивать эту емкость вручную, вызывая метод `ensureCapacity()`. Это может потребоваться, если вы заранее знаете, что собираетесь сохранить в коллекции намного больше элементов, чем она содержит в данный момент. Увеличивая емкость однажды, в начале работы, вы тем самым предотвращаете несколько дополнительных распределений памяти позднее. Поскольку перераспределения памяти — дорогостоящие операции в смысле временных затрат, предотвращение таких излишних операций увеличивает производительность. Сигнатура метода `ensureCapacity()` показана ниже.

```
void ensureCapacity(int емкость)
```

Здесь *емкость* задает новую минимальную емкость коллекции.

И наоборот, когда вы хотите уменьшить размер массива объектов, лежащего в основе объекта класса `ArrayList`, до текущего реального количества хранимых объектов, вызовите метод `trimToSize()`.

```
void trimToSize()
```

Получение массива из объекта класса ArrayList

При работе с классом ArrayList иногда необходимо получить обыкновенный массив, содержащий все элементы списка. Это можно сделать, вызвав метод toArray(), который определен в интерфейсе Collection.

Существует несколько причин, по которым вам может понадобиться преобразовать коллекцию в массив.

- Для ускорения выполнения некоторых операций.
- Для передачи массива в качестве параметра методам, не перегруженным для непосредственного использования коллекций.
- Для интеграции нового кода, основанного на коллекциях с унаследованным кодом, который не понимает коллекций.

Независимо от причины, преобразование объекта класса ArrayList в массив — задача тривиальная.

Как объяснялось ранее, существует две версии метода toArray(), которые показаны ниже.

```
Object[] toArray()
<T> T[] toArray(T массив[])
```

Первая версия возвращает массив объектов класса Object, вторая — массив элементов, имеющих тип T. Обычно вторая форма удобнее, поскольку возвращает правильный тип массива. В следующей программе демонстрируется их применение.

```
// Преобразование ArrayList в массив.
import java.util.*;

class ArrayListToArray {
    public static void main(String args[]) {
        // Создать массив-список.
        ArrayList<Integer> al = new ArrayList<Integer>();

        // Добавить элементы в массив-список.
        al.add(1);
        al.add(2);
        al.add(3);
        al.add(4);

        System.out.println("Содержимое al: " + al);

        // Получить массив.
        Integer ia[] = new Integer[al.size()];
        ia = al.toArray(ia);

        int sum = 0;

        // Суммировать массив.
        for(int i : ia) sum += i;

        System.out.println("Сумма: " + sum);
    }
}
```

Вывод программы будет выглядеть так.

```
Содержимое al: [1, 2, 3, 4]
Сумма: 10
```

Программа начинается с создания коллекции целых чисел. Затем вызывается метод `toArray()`, и получается массив элементов класса `Integer`. Далее содержимое массива суммируется в цикле `for` вида "for-each".

Есть еще кое-что интересное в этой программе. Как вы знаете, коллекции могут содержать только ссылки, а не значения элементарных типов. Однако автоматическая упаковка позволяет передавать методу `add()` значения типа `int`, без необходимости помещать их в оболочку `Integer`, как это демонстрируется в программе. Это осуществляет автоматическая упаковка. Таким образом, она ощутимо облегчает сохранение в коллекциях значений элементарных типов.

Класс `LinkedList`

Этот класс расширяет класс `AbstractSequentalList` и реализует интерфейсы `List`, `Dequeue` и `Queue`. Он представляет структуру данных связного списка. Класс `LinkedList` — это обобщенный класс со следующим объявлением.

```
class LinkedList<E>
```

Здесь *E* указывает тип сохраняемых в списке объектов. Класс `LinkedList` имеет следующие два конструктора.

```
LinkedList()
LinkedList(Collection<? extends E> c)
```

Первый конструктор создает пустой связный список. Второй — создает связный список и инициирует его содержимым коллекции *c*.

Поскольку класс `LinkedList` реализует интерфейс `Deque`, вы имеете доступ к методам, определенным в интерфейсе `Deque`. Например, чтобы добавить элементы в начало списка, вы можете использовать методы `addFirst()` или `offerFirst()`. Для добавления элементов в конец списка применяйте методы `addLast()` или `offerLast()`. Чтобы получить первый элемент, используйте методы `getLast()` или `peekLast()`. Чтобы удалить первый элемент, вызывайте методы `removeFirst()` или `pollFirst()`. Для удаления последнего элемента используются методы `removeLast()` или `pollLast()`.

В следующей программе иллюстрируется использование класса `LinkedList`.

```
// Демонстрация применения 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("ll после удаления первого и последнего: "+
                           ll);

        // Получить и присвоить значение.
        String val = ll.get(2);
        ll.set(2, val + " Изменен");

        System.out.println("ll после изменения: " + ll);
    }
```

Ниже показан вывод этой программы.

```
Исходное содержимое ll: [A, A2, F, B, D, E, C, Z]
Содержимое ll после удаления: [A, A2, D, E, C, Z]
ll после удаления первого и последнего: [A2, D, E, C]
ll после изменения: [A2, D, E Changed, C]
```

Поскольку класс `LinkedList` реализует интерфейс `List`, вызов метода `add(E)` добавляет элементы в конец списка, как это делает и метод `addLast()`. Чтобы вставить элементы в определенное место, используйте форму `add(int, E)`, как демонстрирует в примере вызов метода `add(1, "A2")`.

Обратите внимание на то, как изменяется третий элемент в связном списке `ll` с помощью методов `get()` и `set()`. Чтобы получить текущее значение элемента, методу `get()` передается индекс позиции, в которой расположен элемент. Чтобы присвоить новое значение элементу в этой позиции, методу `set()` передается соответствующий индекс и новое значение.

Класс `HashSet`

Класс `HashSet` расширяет класс `AbstractSet` и реализует интерфейс `Set`. Он создает коллекцию, которая использует для хранения хеш-таблицу. Класс `HashSet` — это обобщенный класс, имеющий следующее объявление.

```
class HashSet<E>
```

Здесь *E* указывает тип объектов, которые будут храниться в наборе.

Как большинство читателей, вероятно, знают, хеш-таблица хранит информацию, используя так называемый механизм *хеширования*, в котором содержимое ключа используется для определения уникального значения, называемого *хеш-кодом*. Этот хеш-код затем применяется в качестве индекса, с которым ассоциируются данные, доступные по этому ключу. Преобразование ключа в хеш-код выполняется автоматически — вы никогда не увидите самого хеш-кода. Также ваш код не может напрямую индексировать хеш-таблицу. Выгода от хеширования состоит в том, что оно обеспечивает константное время выполнения методов `add()`, `contains()`, `remove()` и `size()`, даже для больших наборов.

Определены следующие конструкторы.

```
HashSet()
HashSet(Collection<? extends E> c)
HashSet(int емкость)
HashSet(int емкость, float коэффЗаполнения)
```

Первая форма конструктора создает хеш-набор по умолчанию. Вторая форма инициирует его содержимым коллекции *c*. Третья форма устанавливает емкость хеш-набора равной *емкость*. (Емкость по умолчанию — 16.) Четвертая форма инициирует и емкость, и *коэффициент заполнения* (fill ratio), называемый также *емкостью загрузки* (load capacity) из аргументов конструктора. Коэффициент заполнения должен быть в пределах от 0,0 до 1,0, и это определяет, насколько заполненным должен быть хеш-набор, прежде чем будет выполнено изменение его размера. Точнее говоря, когда количество элементов становится больше емкости хеш-набора, умноженной на коэффициент заполнения, такой хеш-набор увеличивается. В конструкторах, которые не принимают параметр коэффициента заполнения, принимается значение 0,75.

Класс `HashSet` не определяет никаких дополнительных методов, помимо представленных его суперклассами и интерфейсами.

Важно отметить, что класс `HashSet` не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не порождает сортированных наборов. Если вам нужны сортированные наборы, то лучшим выбором может быть другой тип коллекций, такой как класс `TreeSet`.

Ниже показан пример применения класса `HashSet`.

```
// Демонстрация применения HashSet.
import java.util.*;

class HashSetDemo {
    public static void main(String args[]) {
        // Создать хеш-набор.
        HashSet<String> hs = new HashSet<String>();

        // Добавить элементы в хеш-набор.
        hs.add("B");
        hs.add("A");
        hs.add("D");
        hs.add("E");
        hs.add("C");
        hs.add("F");

        System.out.println(hs);
    }
}
```

Вывод этой программы выглядит следующим образом.

```
[D, E, F, A, B, C]
```

Как объяснялось, элементы не сохраняются в отсортированном порядке, поэтому порядок их вывода может варьироваться.

Класс `LinkedHashSet`

Класс `LinkedHashSet` расширяет класс `HashSet`, не добавляя никаких новых методов. Это обобщенный класс со следующим объявлением.

```
class LinkedHashSet<E>
```

Здесь *E* указывает тип объектов, которые будут храниться в наборе. Конструкторы этого класса аналогичны конструкторам класса `HashSet`.

Класс `LinkedHashSet` поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор. То есть, когда идет перебор объекта класса `LinkedHashSet` с применением итератора, элементы извлекаются в том порядке, в каком они были

вставлены. Это также тот порядок, в котором они будут возвращены методом toString() объекта класса LinkedHashSet. Чтобы увидеть эффект от применения класса LinkedHashSet, попробуйте подставить его в предыдущем примере вместо класса HashSet. Вывод программы после этого будет выглядеть так.

```
[B, A, D, E, C, F]
```

Это отражает порядок, в каком элементы вставлялись в набор.

Класс TreeSet

Класс TreeSet расширяет класс AbstractSet и реализует интерфейс NavigableSet. Он создает коллекцию, которая для хранения элементов применяет дерево. Объекты сохраняются в отсортированном порядке по возрастанию. Время доступа и извлечения элементов достаточно мало, что делает класс TreeSet блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена. Класс TreeSet — это обобщенный класс со следующим объявлением.

```
class TreeSet<E>
```

Здесь *E* указывает тип объектов, которые будут храниться в наборе.

Класс TreeSet имеет следующие конструкторы.

```
TreeSet()
TreeSet(Collection<? extends E> c)
TreeSet(Comparator<? super E> компаратор)
TreeSet(SortedSet<E> ss)
```

Первая форма создает пустой набор-дерево, который будет сортировать элементы в естественном порядке возрастания. Вторая форма создает набор-дерево, содержащий элементы коллекции *c*. Третья форма создает пустой набор-дерево, элементы в котором будут отсортированы компаратором, указанным в параметре *компаратор*. (Компараторы рассматриваются далее в настоящей главе.) Четвертая форма создает набор-дерево, содержащий элементы из *ss*. Ниже показан пример использования класса TreeSet.

```
 / Демонстрация TreeSet.
import java.util.*;

class TreeSetDemo {
    public static void main(String args[]) {
        // Создать TreeSet.
        TreeSet<String> ts = new TreeSet<String>();

        // Добавить элементы в TreeSet.
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");

        System.out.println(ts);
    }
}
```

Вот как выглядит вывод этой программы.

```
[A, B, C, D, E, F]
```

Как уже объяснялось, поскольку класс TreeSet сохраняет элементы дерева, они автоматически располагаются в отсортированном порядке, что и подтверждает вывод программы.

Поскольку класс TreeSet реализует интерфейс NavigableSet (добавленный в Java SE 6), вы можете использовать методы, определенные в интерфейсе NavigableSet, для извлечения элементов класса TreeSet. Например, предположим, что в предыдущую программу был добавлен следующий оператор, который использует метод subSet() для получения набора ts, содержащего элементы между C (включительно) и F (исключительно). Затем отображается результирующий набор.

```
System.out.println(ts.subSet("C", "F"));
```

Вывод этого оператора показан ниже.

```
[C, D, E]
```

При желании вы можете поэкспериментировать с другими методами, определенными в интерфейсе NavigableSet.

Класс PriorityQueue

Класс PriorityQueue расширяет класс AbstractQueue и реализует интерфейс Queue. Он создает очередь с приоритетами на базе компаратора очереди. Класс PriorityQueue является обобщенным классом со следующим объявлением.

```
class PriorityQueue<E>
```

Здесь *E* указывает тип объектов, которые будут храниться в очереди. Объект класса PriorityQueue является динамической коллекцией и при необходимости может увеличиваться.

Класс PriorityQueue определяет шесть конструкторов.

```
PriorityQueue()
PriorityQueue(int емкость)
PriorityQueue(int емкость, Comparator<? super E> компаратор)
PriorityQueue(Collection<? extends E> c)
PriorityQueue(PriorityQueue<? extends E> c)
PriorityQueue(SortedSet<? extends E> c)
```

Первый конструктор создает пустую очередь. Его начальная емкость равна 11. Второй конструктор создает очередь с заданной начальной емкостью. Третий конструктор создает очередь заданной емкости с заданным компаратором. Последние три конструктора создают очереди, инициированные элементами коллекций, переданных в параметре *c*. Во всех случаях по мере добавления элементов емкость автоматически растет.

Если при создании объекта класса PriorityQueue никакой компаратор не указан, то применяется стандартный компаратор для того типа данных, который сохраняется в очереди. Стандартный компаратор размещает элементы очереди в порядке возрастания. Таким образом, в начале (голове) очереди будет находиться наименьшее значение. Однако, предоставляя собственный компаратор, вы можете задать другую схему сортировки элементов. Например, когда сохраняются элементы, включающие временную метку, вы можете ввести приоритеты в очередь таким образом, чтобы самые "старые" элементы располагались в начале очереди.

Вы можете получить ссылку на компаратор, используемый объектом класса PriorityQueue, вызвав его метод comparator().

```
Comparator<? super E> comparator()
```

Возвращает компаратор. Если в данной очереди применяется естественный порядок сортировки, возвращается значение `null`. Одно предупреждение: несмотря на то что вы можете перебрать элементы объекта класса `PriorityQueue`, применяя итератор, порядок итерации не определен. Чтобы правильно использовать класс `PriorityQueue`, следует вызывать такие методы, как `offer()` и `poll()`.

Класс `ArrayDeque`

Класс `ArrayDeque` расширяет класс `AbstractCollection` и реализует интерфейс `Deque`. Он не добавляет собственных методов. Класс `ArrayDeque` создает динамический массив, не имеющий ограничений емкости. (Интерфейс `Deque` поддерживает реализации с ограниченной емкостью, но не накладывает такого ограничения.) Класс `ArrayDeque` — это обобщенный класс со следующим объявлением.

```
class ArrayDeque<E>
```

Здесь *E* определяет тип объекта, сохраняемого в коллекции. Класс `ArrayDeque` определяет следующие конструкторы.

```
ArrayDeque()
ArrayDeque(int размер)
ArrayDeque(Collection<? extends E> c)
```

Первый конструктор создает пустую двунаправленную очередь. Ее начальная емкость — 16 элементов. Второй конструктор создает двунаправленную очередь с указанной емкостью. Третий конструктор создает двунаправленную очередь, инициализируемую коллекцией, переданной в параметре *c*. Во всех случаях, по мере необходимости, емкость увеличивается при добавлении новых элементов в двунаправленную очередь.

Приведенная ниже программа демонстрирует применение класса `ArrayDeque` для организации стека.

```
// Демонстрация применения ArrayDeque.
import java.util.*;

class ArrayDequeDemo {
    public static void main(String args[]) {
        // Создать двухстороннюю очередь.
        ArrayDeque<String> adq = new ArrayDeque<String>();

        // Использование ArrayDeque в виде стека.
        adq.push("A");
        adq.push("B");
        adq.push("D");
        adq.push("E");
        adq.push("F");

        System.out.print("Выталкиваем из стека: ");

        while(adq.peek() != null)
            System.out.print(adq.pop() + " ");

        System.out.println();
    }
}
```

Вывод этой программы выглядит так.

```
Выталкиваем из стека: F E D B A
```

Класс EnumSet

Класс EnumSet расширяет класс AbstractSet и реализует интерфейс Set. Он создает коллекцию, которая предназначена для применения с ключами типа enum. Это обобщенный класс со следующим объявлением.

```
class EnumSet<E extends Enum<E>>
```

Здесь *E* определяет элементы. Отметим, что класс *E* должен расширять класс Enum<*E*>, а это накладывает требование, что элементы должны относиться к определенному типу enum.

Класс EnumSet не определяет конструкторов. Вместо этого для создания объектов он использует методы фабрики, перечисленные в табл. 17.9. Обратите внимание на то, что метод of() перегружен множество раз. Это делается из соображений эффективности. Передача известного количества аргументов может работать быстрее, чем применение параметра с переменным количеством аргументов, когда количество аргументов не велико.

Таблица 17.9. Методы, определенные в классе EnumSet

Метод	Описание
`static <E extends Enum<E>> EnumSet<E> allOf(Class<E> t)`	Создает объект класса EnumSet, который содержит элементы перечисления, указанные в параметре *t*
`static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> e)`	Создает объект класса EnumSet, который дополняет элементы, отсутствующие в *e*
`static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> c)`	Создает объект класса EnumSet, содержащий элементы из набора *c*
`static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)`	Создает объект класса EnumSet, содержащий элементы из набора *c*
`static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> t)`	Создает объект класса EnumSet, содержащий элементы, которые не входят в перечисление, заданное *t*, которое по определению является пустым набором
`static <E extends Enum<E>> EnumSet<E> of(E v, E ... перемколарг)`	Создает объект класса EnumSet, содержащий элементы *v* и нуль или более дополнительных перечислимых значений
`static <E extends Enum<E>> EnumSet<E> of(E v)`	Создает объект класса EnumSet, содержащий элементы *v*
`static <E extends Enum<E>> EnumSet<E> of(E v1, E v2)`	Создает объект класса EnumSet, содержащий элементы *v1* и *v2*
`static <E extends Enum<E>> EnumSet<E> of(E v1, E v2, E v3)`	Создает объект класса EnumSet, содержащий элементы от *v1* до *v3*
`static <E extends Enum<E>> EnumSet<E> of(E v1, E v2, E v3, E v4)`	Создает объект класса EnumSet, содержащий элементы от *v1* до *v4*
`static <E extends Enum<E>> EnumSet<E> of(E v1, E v2, E v3, E v4, E v5)`	Создает объект класса EnumSet, содержащий элементы от *v1* до *v5*
`static <E extends Enum<E>> EnumSet<E> range(E начало, E конец)`	Создает объект класса EnumSet, содержащий элементы в диапазоне, заданном *начало* и *конец*

Доступ к коллекциям через итератор

Зачастую необходимо перебрать все элементы коллекции. Например, необходимо отобразить каждый элемент коллекции. Один из способов сделать это — использовать итератор, представляющий собой объект, реализующий один из двух интерфейсов — `Iterator` либо `ListIterator`. Интерфейс `Iterator` позволяет организовать цикл для перебора коллекции, получая либо удаляя элементы. Интерфейс `ListIterator` расширяет интерфейс `Iterator` для обеспечения двунаправленного прохода по списку и модификации элементов. Интерфейсы `Iterator` и `ListIterator` — это обобщенные интерфейсы, объявленные так, как показано ниже.

```
interface Iterator<E>
interface ListIterator<E>
```

Здесь *E* представляет собой тип перебираемых объектов. Интерфейс `Iterator` объявляет методы, перечисленные в табл. 17.10. Методы, объявленные в интерфейсе `ListIterator`, показаны в табл. 17.11.

Таблица 17.10. Методы, определенные в интерфейсе `Iterator`

Метод	Описание
`boolean hasNext()`	Возвращает значение `true`, если есть еще элементы. В противном случае возвращает значение `false`
`E next()`	Возвращает следующий элемент. Передает исключение `NoSuchElementException`, если больше нет элементов
`void remove()`	Удаляет текущий элемент. Передает исключение `IllegalStateException`, если предпринимается попытка вызвать метод `remove()`, которому не предшествовал вызов метода `next()`

Таблица 17.11. Методы, определенные в интерфейсе `ListIterator`

Метод	Описание
`void add(E` *объект*`)`	Вставляет *объект* перед элементом, который должен быть возвращен следующим вызовом метода `next()`
`boolean hasNext()`	Возвращает значение `true`, если есть следующий элемент. В противном случае возвращает значение `false`
`boolean hasPrevious()`	Возвращает значение `true`, если есть предыдущий элемент. В противном случае возвращает значение `false`
`E next()`	Возвращает следующий элемент. Если следующего нет, передается исключение `NoSuchElementException`
`int nextIndex()`	Возвращает индекс следующего элемента. Если следующего нет, возвращается размер списка
`E previous()`	Возвращает предыдущий элемент. Если предыдущего нет, передается исключение `NoSuchElementException`
`int previousIndex()`	Возвращает индекс предыдущего элемента. Если предыдущего нет, возвращается значение `-1`
`void remove()`	Удаляет текущий элемент из списка. Если метод `remove()` вызван до метода `next()` или `previous()`, передается исключение `IllegalStateException`
`void set(E` *объект*`)`	Присваивает *объект* текущему элементу. Это элемент, возвращенный последним вызовом метода `next()` или `previous()`

В обоих случаях операции, которые модифицируют лежащую в основе коллекцию, необязательны. Например, метод `remove()` передаст исключение `UnsupportedOperationException`, будучи примененным к коллекции, доступной только для чтения. Возможны также и другие исключения.

Использование интерфейса `Iterator`

Прежде чем обратиться к коллекции через итератор, следует получить его. Каждый класс коллекций предлагает метод `iterator()`, который возвращает итератор на начало коллекции. Используя объект итератора, вы можете получить доступ к каждому элементу коллекции — одному за другим. В общем случае, применение итератора для перебора содержимого коллекции сводится к выполнению следующих действий.

1. Установить итератор на начало коллекции, получив его от метода `iterator()` коллекции.
2. Организовать цикл, вызывающий метод `hasNext()`. Выполнять перебор до тех пор, пока метод `hasNext()` возвращает значение `true`.
3. Внутри цикла получать каждый элемент, вызывая метод `next()`.

Для коллекций, реализующих интерфейс `List`, вы также можете получить итератор, вызывая метод `listIterator()`. Как уже объяснялось, итератор списка обеспечивает доступ к элементам коллекции, как в прямом, так и обратном направлении, а также позволяет модифицировать элементы. Во всем остальном интерфейс `ListIterator` применяется так же, как интерфейс `Iterator`.

В следующем примере выполняются все перечисленные действия с демонстрацией обоих интерфейсов — `Iterator` и `ListIterator`. Здесь используется объект класса `ArrayList`, но общие принципы применимы к коллекциям любого типа. Конечно, интерфейс `ListIterator` доступен только тем коллекциям, которые реализуют интерфейс `List`.

```
// Демонстрация применения итераторов.
import java.util.*;

class IteratorDemo {
    public static void main(String args[]) {
        // Создать массив-список.
        ArrayList<String> al = new ArrayList<String>();

        // Добавить элементы в массив-список.
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");

        // Использовать итераторы 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 текущий объект 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();

        // Теперь отображаем 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();
    }
}
```

Ниже показан вывод этой программы.

```
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 то, как список отображается в обратном порядке. После того как список модифицирован, итератор `litr` указывает на конец списка. (Помните, что метод `litr.hasNext()` возвращает значение `false`, когда достигнут конец списка.) Чтобы перебрать список в обратном порядке, программа продолжает использовать итератор `litr`, но на этот раз она проверяет, существует ли предшествующий элемент. Пока она это делает, извлеченный элемент отображается.

Версия "for-each" цикла `for` как альтернатива итераторам

Если вам не нужно модифицировать содержимое коллекции либо извлекать элементы в обратном порядке, в этом случае версия "for-each" цикла `for` может оказаться более удобной альтернативой итераторам. Вспомните, что цикл `for` может перебирать любую коллекцию объектов, реализующую интерфейс `Iterable`. Поскольку все классы коллекций реализуют этот интерфейс, ими можно оперировать с помощью цикла `for`.

В следующем примере цикл `for` используется для суммирования содержимого коллекции.

```
// Применение цикла "for-each" для перебора элементов коллекции.
import java.util.*;

class ForEachDemo {
    public static void main(String args[]) {
        // Создать массив-список для целых чисел.
        ArrayList<Integer> vals = new ArrayList<Integer>();

        // Добавить значения в массив-список.
```

```
        vals.add(1);
        vals.add(2);
        vals.add(3);
        vals.add(4);
        vals.add(5);

        // Использовать цикл для отображения значений.
        System.out.print("Исходное содержимое vals: ");
        for(int v : vals)
            System.out.print(v + " ");

        System.out.println();

        // Суммирование значений в цикле for.
        int sum = 0;
        for(int v : vals)
            sum += v;

        System.out.println("Сумма значений: " + sum);
    }
}
```

Вывод этой программы показан ниже.

```
Исходное содержимое vals: 1 2 3 4 5
Сумма значений: 15
```

Как видите, цикл `for` существенно проще и короче, чем подход на базе итераторов. Однако он может применяться для построения цикла перебора элементов коллекции только в прямом направлении, и вы не можете модифицировать элементы коллекции.

Использование пользовательских классов в коллекциях

Во всех приведенных ранее примерах в целях простоты в коллекциях сохранялись объекты встроенных классов, таких как `String` или `Integer`. Конечно, коллекции не ограничиваются сохранением встроенных объектов. Как раз наоборот. Мощность коллекций в том, что они могут хранить любой тип объектов, включая объекты классов, которые создаете вы сами. Например, рассмотрим следующий пример, в котором класс `LinkedList` используется для сохранения 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 список почтовых адресов.
        for(Address element : ml)
            System.out.println(element + "\n");

        System.out.println();
    }
}
```

Вывод этой программы приведен ниже.

```
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 пользовательских классов в коллекции, есть еще кое-что, заслуживающее внимания, — эта программа достаточно короткая. Когда вы оцените, что для сохранения, извлечения и обработки почтовых адресов понадобилось всего около 50 строк кода, станет очевидной мощь инфраструктуры коллекций. Как большинство пользователей знают, если все эти функциональные возможности запрограммировать вручную, программа станет в несколько раз длиннее. Коллекции предлагают готовые решения для широкого диапазона программистских задач. Их следует использовать всякий раз, когда ситуация позволяет.

Интерфейс RandomAccess

Этот интерфейс не имеет членов. Однако, реализуя его, коллекция сообщает о том, что поддерживает эффективный случайный доступ к своим элементам. Несмотря на то что коллекция может поддерживать случайный доступ, он может быть не слишком эффективным. Проверяя интерфейс `RandomAccess`, клиентский код может определять во время выполнения, допускает ли конкретная коллекция не-

которые типы операций случайного доступа — особенно, насколько они применимы к большим коллекциям. (Вы можете использовать оператор `instanceof` для определения того, реализует ли класс данный интерфейс.) Интерфейс `RandomAccess` реализуется классом `ArrayList` и, среди прочих, унаследованным классом `Vector`.

Работа с картами

Карта (map) — это объект, который сохраняет ассоциации между ключами и значениями, или пары "ключ-значение". По заданному ключу вы можете найти его значение. И ключи, и значения являются объектами. Ключи могут быть уникальными, а значения могут дублироваться. Одни карты допускают пустые ключи и пустые значения, другие — нет.

Имеется один ключевой момент относительно карт, который важно упомянуть: они не реализуют интерфейс `Iterable`. Это означает, что вы не можете перебирать карту, используя форму "for-each" цикла `for`. Более того, вы не можете получить итератор карты. Однако, как вскоре будет показано, можно получить представление карты в виде коллекции, которое допускает использование и цикла, и итераторов.

Интерфейсы карт

Поскольку интерфейсы карт определяют их характер и природу, обсуждение начнем с них. Интерфейсы, перечисленные в табл. 17.12, поддерживают карты.

Таблица 17.12. Интерфейсы, которые поддерживают карты

Интерфейс	Описание
`Map`	Отображает уникальные ключи на значения
`Map.Entry`	Описывает элемент карты (пару "ключ-значение"). Это — вложенный класс интерфейса `Map`
`NavigableMap`	Расширяет интерфейс `SortedMap` для обработки извлечения элементов на основе поиска по ближайшему соответствию
`SortedMap`	Расширяет `Map` таким образом, что ключи располагаются в порядке по возрастанию

Ниже каждый из этих интерфейсов рассматривается более подробно.

Интерфейс `Map`

Интерфейс `Map` соотносит уникальные ключи со значениями. *Ключ* — это объект, который вы используете для последующего извлечения данных. Задавая ключ и значение, вы можете помещать *значения* в объект карты. После того как это значение сохранено, вы можете получить его по ключу. Интерфейс `Map` — это обобщенный интерфейс, объявленный так, как показано ниже.

```
interface Map<K, V>
```

Здесь *K* указывает тип ключей, а *V* — тип хранимых значений.

Методы, объявленные в интерфейсе `Map`, собраны в табл. 17.13. Несколько методов передают исключение `ClassCastException`, когда объект оказывается несовместимым с элементами карты. Исключение `NullPointerException` передается при попытке использовать пустой объект, когда данная карта этого не допускает. Исключение `UnsupportedOperationException` передается при попытке изменить немодифицируемую карту.

Таблица 17.13. Методы, определенные в интерфейсе Map

Метод	Описание
`void clear()`	Удаляет все пары "ключ-значение" из вызывающей карты
`boolean containsKey(Object k)`	Возвращает значение `true`, если вызывающая карта содержит ключ *k*. В противном случае возвращает значение `false`
`boolean containsValue(Object v)`	Возвращает значение `true`, если вызывающая карта содержит значение *v*. В противном случае возвращает значение `false`
`Set<Map.Entry<K, V>> entrySet()`	Возвращает набор, содержащий все значения карты. Набор содержит объекты интерфейса `Map.Entry`. То есть этот метод представляет карту в виде набора
`boolean equals(Object объект)`	Возвращает значение `true`, если *объект* — это карта, содержащая одинаковые значения. В противном случае возвращает значение `false`
`V get(Object k)`	Возвращает значение, ассоциированное с ключом *k*. Возвращает значение `null`, если ключ не найден
`int hashCode()`	Возвращает хеш-код вызывающей карты
`boolean isEmpty()`	Возвращает значение `true`, если вызывающая карта пуста. В противном случае возвращает значение `false`
`Set<K> keySet()`	Возвращает набор, содержащий ключи вызывающей карты. Этот метод представляет ключи вызывающей карты в виде набора
`V put(K k, V v)`	Помещает элемент в вызывающую карту, перезаписывая любое предшествующее значение, ассоциированное с ключом. Ключ и значение — это *k* и *v* соответственно. Возвращает значение `null`, если ключ ранее не существовал. В противном случае возвращается предыдущее значение, связанное с ключом
`void putAll(Map<? extends K, ? extends V> m)`	Помещает все значения из *m* в карту
`V remove(Object k)`	Удаляет элемент, ключ которого равен *k*
`int size()`	Возвращает количество пар "ключ-значение" в карте
`Collection<V> values()`	Возвращает коллекцию, содержащую значения карты. Этот метод представляет содержащиеся в карте значения в виде коллекции

Карты вращаются вокруг двух основных операторов: `get()` и `put()`. Чтобы поместить значение в карту, используйте метод `put()`, указав ключ и значение. Чтобы получить значение, вызовите метод `get()`, передавая ключ в качестве аргумента. Значение будет возвращено.

Как упоминалось ранее, несмотря на то что карты являются частью инфраструктуры коллекций, сами по себе они не реализуют интерфейс `Collection`. Однако вы можете получить представление карт в виде коллекций. Для этого можно воспользоваться методом `entrySet()`, который возвращает набор, содержащий элементы карты. Чтобы получить коллекционное представление ключей, используйте метод `keySet()`. Чтобы получить коллекционное представление значений, используйте метод `values()`. Коллекционные представления — это средства, которыми карты интегрируются в большую инфраструктуру коллекций.

Интерфейс SortedMap

Этот интерфейс расширяет интерфейс Map. Он гарантирует, что элементы размещаются в возрастающем порядке значений ключей. Интерфейс SortedMap — это обобщенный интерфейс, объявленный так, как показано ниже.

```
interface SortedMap<K, V>
```

Здесь *K* указывает тип ключей, а *V* — тип хранимых значений.

Методы, объявленные в интерфейсе SortedMap, собраны в табл. 17.14. Некоторые методы передают исключение NoSuchElementException, когда вызывающая карта пуста. Исключение ClassCastException передается при несовместимости объекта с элементами, хранящимися в карте. Исключение NullPointerException передается при попытке использовать пустой объект, когда пустые объекты в данной карте недопустимы. Исключение IllegalArgumentException передается при использовании неверного аргумента.

Таблица 17.14. Методы, определенные в интерфейсе SortedMap

Метод	Значение
`Comparator<? super K> comparator()`	Возвращает компаратор вызывающей отсортированной карты. Если карта использует естественный порядок, возвращается значение null
`K firstKey()`	Возвращает первый ключ вызывающей карты
`SortedMap<K, V> headMap(K конец)`	Возвращает отсортированную карту, содержащую те элементы вызывающей карты, ключ которых меньше *конец*
`K lastKey()`	Возвращает последний ключ вызывающей карты
`SortedMap<K, V> subMap(K начало, K конец)`	Возвращает карту, содержащую элементы вызывающей карты, ключ которых больше или равен *начало* и меньше *конец*
`SortedMap<K, V> tailMap(K начало)`	Возвращает отсортированную карту, содержащую те элементы вызывающей карты, ключ которых больше *начало*

Отсортированные карты обеспечивают очень эффективное манипулирование *подкартами* (другими словами, подмножествами карты). Чтобы получить подкарту, используйте методы headMap(), tailMap() или subMap(). Подкарта, возвращенная этими методами, поддерживается вызывающей картой. При изменении одной меняется другая. Чтобы получить первый ключ набора, вызывайте метод firstKey(). Чтобы получить последний ключ — применяйте метод lastKey().

Интерфейс NavigableMap

Интерфейс NavigableMap расширяет интерфейс SortedMap и определяет поведение карты, поддерживающей извлечение элементов на основе ближайшего соответствия заданному ключу или ключам. Интерфейс NavigableMap — это обобщенный интерфейс со следующим объявлением.

```
interface NavigableMap<K, V>
```

Здесь *K* определяет тип ключей, а *V* — тип значений, ассоциированных с ключами. В дополнение к методам, унаследованным от интерфейса SortedMap, интерфейс NavigableMap добавляет методы, перечисленные в табл. 17.15. Некоторые методы

передают исключение `ClassCastException`, когда объект несовместим с ключами карты. Исключение `NullPointerException` передается при попытке использования пустого объекта, когда пустые ключи не допускаются в наборе. Исключение `IllegalArgumentException` передается при неправильном аргументе.

Таблица 17.15. Методы, определенные в интерфейсе NavigableMap

Метод	Значение
`Map.Entry<K, V> ceilingEntry(K` *объект*`)`	Выполняет поиск в карте наименьшего ключа *k*, такого, что *k*>=*объект*. Если такой ключ найден, возвращается соответствующее ему значение, в противном случае возвращается значение `null`
`K ceilingKey(K` *объект*`)`	Выполняет поиск в карте наименьшего ключа *k*, такого, что *k*>=*объект*. Если такой ключ найден, он возвращается, в противном случае возвращается значение `null`
`NavigableSet<K> descendingKeySet()`	Возвращает объект интерфейса `NavigableSet`, содержащий ключи вызывающей карты в обратном порядке. То есть он возвращает обратное представление ключей. Результирующий набор основан на вызывающей карте
`NavigableMap<K, V> descendingMap()`	Возвращает объект интерфейса `NavigableSet`, обратный вызывающей карте
`Map.Entry<K, V> firstEntry()`	Возвращает первое вхождение в карте. Это будет вхождение с минимальным ключом. Результирующий набор основан на вызывающей карте
`Map.Entry<K, V> floorEntry(K` *объект*`)`	Выполняет поиск в карте наибольшего ключа *k*, такого, что *k*<=*объект*. Если такой ключ найден, возвращается соответствующее ему значение, в противном случае возвращается значение `null`
`K floorKey(K` *объект*`)`	Выполняет поиск в карте наибольшего ключа *k*, такого, что *k*<=*объект*. Если такой ключ найден, он возвращается, в противном случае возвращается значение `null`
`NavigableMap<K, V> headMap(K` *верхнГраница*`, boolean` *включать*`)`	Возвращает объект интерфейса `NavigableSet`, включающий все вхождения вызывающей карты, имеющей ключи, меньшие *верхнГраница*. Если параметр *включать* содержит значение `true`, то элемент, равный *верхнГраница*, включается. Результирующий набор основан на вызывающей карте
`Map.Entry<K, V> higherEntry(K` *объект*`)`	Выполняет поиск в наборе наибольшего ключа, такого, что *k*>*объект*. Если такой ключ найден, возвращается соответствующее ему значение. В противном случае возвращается значение `null`
`K higherKey(K` *объект*`)`	Выполняет поиск в наборе наибольшего ключа, такого, что *k*>*объект*. Если такой ключ найден, он возвращается. В противном случае возвращается значение `null`
`Map.Entry<K, V> lastEntry()`	Возвращает последнее вхождение в карте. Это будет значение с наибольшим ключом
`Map.Entry<K, V> lowerEntry(K` *объект*`)`	Выполняет поиск в наборе наибольшего ключа, такого, что *k*<*объект*. Если такой ключ найден, возвращается соответствующее ему значение. В противном случае возвращается значение `null`

Окончание табл. 17.15

Метод	Значение
`K lowerKey(K объект)`	Выполняет поиск в наборе наибольшего ключа, такого, что *k*<*объект*. Если такой ключ найден, он возвращается. В противном случае возвращается значение `null`
`NavigableSet<K> navigableKeySet()`	Возвращает объект интерфейса `NavigableSet`, содержащий ключи вызывающей карты. Результирующий набор основан на вызывающей карте
`Map.Entry<K, V> pollFirstEntry()`	Возвращает первое вхождение, удаляя его в процессе. Поскольку карта отсортирована, это будет вхождение с наименьшим ключом. При пустой карте возвращается значение `null`
`Map.Entry<K, V> pollLastEntry()`	Возвращает последнее вхождение, удаляя его в процессе. Поскольку карта отсортирована, это будет вхождение с наименьшим ключом. При пустой карте возвращается значение `null`
`NavigableMap<K, V> subMap(K нижнГраница, boolean включатьНижн, K верхнГраница, boolean включатьВерхн)`	Возвращает объект интерфейса `NavigableSet`, включающий все вхождения вызывающей карты, которая имеет ключи, меньшие *верхнГраница* и большие *нижнГраница*. Если параметр *включатьНижн* содержит значение `true`, то элемент, равный *нижнГраница*, включается. Если параметр *включатьВерхн* содержит значение `true`, то элемент, равный *верхнГраница*, включается. Результирующий набор основан на вызывающей карте
`NavigableMap<K, V> tailMap(K нижнГраница, boolean включать)`	Возвращает объект интерфейса `NavigableSet`, включающий все вхождения вызывающей карты, имеющей ключи, большие *нижнГраница*. Если параметр *включать* содержит значение `true`, то элемент, равный *нижнГраница*, включается. Результирующий набор основан на вызывающей карте

Интерфейс `Map.Entry`

Этот интерфейс позволяет работать с элементом карты. Вспомните, что метод `entrySet()`, объявленный в интерфейсе `Map`, возвращает набор, содержащий элементы карты. Каждый элемент этого набора представляет собой объект интерфейса `Map.Entry`. Интерфейс `Map.Entry` является обобщенным и объявлен следующим образом.

```
interface Map.Entry<K, V>
```

Здесь *K* указывает тип ключей, а *V* — тип хранимых значений. В табл. 17.16 перечислены методы, объявленные в интерфейсе `Map.Entry`.

Таблица 17.16. Методы, определенные в интерфейсе `Map.Entry`

Метод	Значение
`boolean equals(Object объект)`	Возвращает значение `true`, если *объект* — это объект интерфейса `Map.Entry`, ключ и значение которого эквивалентны вызывающему объекту
`K getKey()`	Возвращает ключ данного элемента карты
`V getValue()`	Возвращает значение данного элемента карты
`int hashCode()`	Возвращает хеш-код данного элемента карты

Окончание табл. 17.16

Метод	Значение
`V setValue(V v)`	Устанавливает значение данного элемента карты равным *v*. Если *v* не относится к типу, допустимому для данной карты, передается исключение `ClassCastException`. Исключение `IllegalArgumentException` передается при возникновении проблемы с *v*. Исключение `NullPointerException` передается, если параметр *v* содержит значение `null`, а карта не допускает хранения пустых ключей. Исключение `UnsupportedOperationException` передается, если карта не может быть модифицирована

Классы карт

Реализацию интерфейсов карт предлагают несколько классов. Классы, которые могут быть использованы для карт, перечислены в табл. 17.17.

Следует отметить, что класс `AbstractMap` – это суперкласс для всех конкретных реализаций карт.

Класс `WeakHashMap` реализует карту, которая использует "слабые ключи", что позволяет элементу карты быть объектом, подлежащим сбору "мусора", когда его ключ никак не используется. Этот класс подробно здесь не обсуждается. Прочие классы карт описаны ниже.

Таблица 17.17. Классы, которые могут использоваться для карт

Класс	Описание
`AbstractMap`	Реализует большую часть интерфейса `Map`
`EnumMap`	Расширяет класс `AbstractMap` для использования с ключами типа `enum`
`HashMap`	Расширяет класс `AbstractMap` для использования хеш-таблицы
`TreeMap`	Расширяет класс `AbstractMap` для использования дерева
`WeakHashMap`	Расширяет класс `AbstractMap` для использования хеш-таблицы со слабыми ключами
`LinkedHashMap`	Расширяет класс `HashMap`, разрешая перебор в порядке вставки
`IdentityHashMap`	Расширяет класс `AbstractMap` и использует проверку ссылочной эквивалентности при сравнении документов

Класс `HashMap`

Этот класс расширяет класс `AbstractMap` и реализует интерфейс `Map`. Он использует хеш-таблицу для хранения карты, что обеспечивает константное время выполнения методов `get()` и `put()` даже при больших наборах. Класс `HashMap` – это обобщенный класс со следующим объявлением.

```
class HashMap<K, V>
```

Здесь *K* указывает тип ключей, а *V* – тип хранимых значений.

В классе определены следующие конструкторы.

```
HashMap()
HashMap(Map<? extends K, ? extends V> m)
HashMap(int емкость)
HashMap(int емкость, float коэффЗаполнения)
```

Первая форма создает хеш-карту по умолчанию. Вторая форма инициирует хеш-карту элементами *m*. Третья форма инициирует емкость хеш-карты величиной *емкость*. Четвертая форма инициализирует и емкость, и коэффициент заполнения хеш-карты, используя аргументы конструктора. Значение емкости и коэффициента заполнения — то же самое, что и в классе `HashSet`, описанном ранее. Емкость по умолчанию — 16, коэффициент заполнения — 0,75.

Класс `HashMap` реализует интерфейс `Map` и расширяет класс `AbstractMap`. Он не добавляет никаких собственных методов.

Следует отметить, что хеш-карта не гарантирует порядка элементов. Таким образом, порядок, в котором элементы добавляются к хеш-карте, не обязательно соответствует порядку, в котором они читаются итератором. В следующей программе иллюстрируется применение класса `HashMap`. Она соотносит имена вкладчиков с балансовыми счетами. Обратите внимание на то, как получается и используется представление 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("Джон Доу", new Double(3434.34));
        hm.put("Том Смит", new Double(123.22));
        hm.put("Джейн Бейкер", new Double(1378.00));
        hm.put("Тод Холл", new Double(99.22));
        hm.put("Ральф Смит", new Double(-19.08));

        // Получить набор элементов.
        Set<Map.Entry<String, Double>> set = hm.entrySet();

        // Отобразить набор.
        for(Map.Entry<String, Double> me : set) {
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();

        // Добавить 1000 на счет Джона Доу.
        double balance = hm.get("Джон Доу");
        hm.put("Джон Доу ", balance + 1000);
        System.out.println("Новый баланс Джона Доу: " +
                           hm.get("Джон Доу"));
    }
}
```

Вывод этой программы показан здесь (точный порядок может отличаться).

```
Ральф Смит: -19.08
Том Смит: 123.22
Джон Доу: 3434.34
Тод Холл: 99.22
Джейн Бейкер: 1378.0
Новый баланс Джона Доу: 4434.34
```

Программа начинается с создания хеш-карты с последующим соотнесением имен с балансами. Затем содержимое карты соотносится с представлением карты в виде набора, полученного от метода `entrySet()`. Ключи и значения соотносятся при помощи вызовов методов `getKey()` и `getValue()`, которые определены

в интерфейсе Map.Entry. Обратите особое внимание на то, как депозит помещается на счет Джона Доу. Метод put() автоматически замещает любое предварительно существовавшее значение, ассоциированное с указанным ключом, новым значением. Таким образом, после того как счет Джона Доу обновлен, хеш-карта по-прежнему содержит только один счет Джона Доу.

Класс TreeMap

Класс TreeMap расширяет класс AbstractMap и реализует интерфейс NavigableMap. Он создает карту, размещенную в древовидной структуре. Класс TreeMap предлагает эффективный способ хранения пар "ключ-значение" в отсортированном порядке и обеспечивает быстрое извлечение. Следует отметить, что, в отличие от хеш-карты, *карта-дерево* (tree-map) гарантирует, что ее элементы будут отсортированы в порядке возрастания ключей. Класс TreeMap является обобщенным классом со следующим объявлением.

```
class TreeMap<K, V>
```

Здесь *K* указывает тип ключей, а *V* — тип хранимых значений. В классе TreeMap определены следующие конструкторы.

```
TreeMap()
TreeMap(Comparator<? super K> компаратор)
TreeMap(Map<? extends K, ? extends V> m)
TreeMap(SortedMap<K, ? extends V> sm)
```

Первая форма создает пустую карту-дерево, которая будет отсортирована с использованием естественного порядка ключей. Вторая форма создает пустую карту, основанную на дереве, которая будет отсортирована с применением компаратора Comparator *компаратор*. (Компараторы обсуждаются далее в настоящей главе.) Третья форма инициализирует карту-дерево с элементами из *m*, которые будут отсортированы по естественному порядку ключей. Четвертая форма создает карту-дерево с элементами из *sm*, которые будут отсортированы в том же порядке, что и *sm*.

Класс TreeMap не определяет дополнительных методов карты, помимо тех, что имеются в интерфейсе NavigableMap и классе AbstractMap. В следующей программе предыдущий пример переделан для использования класса TreeMap.

```
import java.util.*;

class TreeMapDemo {
    public static void main(String args[]) {
        // Создать карту-дерево.
        TreeMap<String, Double> tm = new TreeMap<String, Double>();

        // Поместить элементы в карту.
        tm.put("Джон Доу", new Double(3434.34));
        tm.put("Том Смит", new Double(123.22));
        tm.put("Джейн Бейкер", new Double(1378.00));
        tm.put("Тод Халл", new Double(99.22));
        tm.put("Ральф Смит", new Double(-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();
```

```
        // Добавить 1000 на счет Джона Доу.
        double balance = tm.get("Джон Доу");
        tm.put("Джон Доу", balance + 1000);
        System.out.println("Новый баланс Джона Доу: " +
        tm.get("Джон Доу"));
    }
}
```

Вот как выглядит вывод программы.

```
Джейн Бейкер: 1378.0
Джон Доу: 3434.34
Ральф Смит: -19.08
Тод Халл: 99.22
Том Смит: 123.22

Текущий баланс Джона Доу: 4434.34
```

Обратите внимание на то, что класс `TreeMap` сортирует ключи. Однако в данном случае они сортируются по имени вместо фамилии. Вы можете изменить это поведение, указав компаратор при создании карты, как уже кратко упоминалось.

Класс `LinkedHashMap`

Класс `LinkedHashMap` расширяет класс `HashMap`. Он создает связный список элементов в карте, расположенных в том порядке, в котором они вставлялись. Это позволяет организовать перебор карты в порядке вставки. То есть, когда происходит итерация по коллекционному представлению объекта класса `LinkedHashMap`, элементы будут возвращаться в том порядке, в котором они вставлялись. Вы также можете создать объект класса `LinkedHashMap`, возвращающий свои элементы в том порядке, в котором к ним в последний раз осуществлялся доступ. Класс `LinkedHashMap` является обобщенным классом, имеющим следующее объявление.

```
class LinkedHashMap<K, V>
```

Здесь *K* указывает тип ключей, а *V* — тип хранимых значений.

Класс `LinkedHashMap` определяет следующие конструкторы.

```
LinkedHashMap()
LinkedHashMap(Map<? extends K, ? extends V> m)
LinkedHashMap(int емкость)
LinkedHashMap(int емкость, float коэффЗаполнения)
LinkedHashMap(int емкость, float коэффЗаполнения, boolean Порядок)
```

Первая форма создает объект класса `LinkedHashMap` по умолчанию. Вторая форма инициализирует объект класса `LinkedHashMap` элементами *m*. Третья форма инициирует емкость. Четвертая форма инициализирует и емкость, и коэффициент заполнения. Смысл этих параметров тот же, что и у класса `HashMap`. Емкость по умолчанию составляет 16, коэффициент заполнения — 0,75. Последняя форма позволяет задать, в каком порядке в связном списке будут размещаться элементы — в порядке вставки или последнего доступа. Если параметр *Порядок* содержит значение `true`, используется порядок доступа, если значение `false` — порядок вставки.

Класс `LinkedHashMap` добавляет только один новый метод к тем, что определены в классе `HashMap`. Этот метод — `removeEldestEntry()`, и он показан ниже.

```
protected boolean removeEldestEntry(Map.Entry<K, V> e)
```

Этот метод вызывается из методов `put()` и `putAll()`. Самый старый элемент передается в *e*. По умолчанию этот метод возвращает значение `false` и не делает ничего. Однако если вы переопределите этот метод, то сможете получить объект класса `LinkedHashMap`, который удаляет самый старый элемент в карте. Чтобы

сделать это, переопределенный метод должен возвращать значение `true`. Чтобы сохранять самый старый элемент, возвращайте значение `false`.

Класс `IdentityHashMap`

Класс `IdentityHashMap` расширяет класс `AbstractMap` и реализует интерфейс `Map`. Он похож на класс `HashMap` всем, за исключением того, что при сравнении элементов использует проверку эквивалентности ссылок. Класс `IdentityHashMap` — это обобщенный класс со следующим объявлением.

```
class IdentityHashMap<K, V>
```

Здесь *K* указывает тип ключей, а *V* — тип хранимых значений. Документация по API явно устанавливает, что класс `IdentityHashMap` не предназначен для общего применения.

Класс `EnumMap`

Класс `EnumMap` расширяет класс `AbstractMap` и реализует интерфейс `Map`. Он специально предназначен для использования с ключами типа `enum`. Это обобщенный класс, имеющий следующее объявление.

```
class EnumMap<K extends Enum<K>, V>
```

Здесь *K* указывает тип ключей, а *V* — тип хранимых значений. Отметим, что класс *K* должен расширять класс `Enum<K>`, а это требует, чтобы ключи были типа `enum`.

В классе `EnumMap` определены следующие конструкторы.

```
EnumMap(Class<K> kТип)
EnumMap(Map<K, ? extends V> m)
EnumMap(EnumMap<K, ? extends V> em)
```

Первый конструктор создает пустой объект класса `EnumMap` для типа *kТип*. Второй создает объект класса `EnumMap`, инициированный значениями из *em*. Класс `EnumMap` не определяет собственных методов.

Компараторы

Классы `TreeSet` и `TreeMap` сохраняют элементы в отсортированном порядке. Однако понятие "порядок сортировки" точно определяет применяемый ими компаратор. По умолчанию эти классы сохраняют элементы, используя то, что в Java называется "естественным порядком", что, как правило, представляет собой тот порядок, которого вы можете ожидать (A перед B, 1 перед 2 и т.д.). Если хотите упорядочить элементы иным образом, то указывайте объект интерфейса `Comparator` при создании набора или карты. Это позволит вам подробно управлять тем, как элементы будут сохраняться в отсортированных коллекциях или картах.

Интерфейс `Comparator` — это обобщенный интерфейс со следующим объявлением.

```
interface Comparator<T>
```

Здесь параметр типа *T* указывает тип сравниваемых объектов.

Интерфейс `Comparator` определяет два метода — `compare()` и `equals()`. Метод `compare()`, представленный ниже, сравнивает два элемента по порядку.

```
int compare(T объект1, T объект2)
```

Здесь *объект1* и *объект2* — это объекты, которые нужно сравнить. Обычно этот метод возвращает значение нуль, если объекты эквивалентны. Он возвращает положительное значение, если *объект1* больше, чем *объект2*, в противном

случае возвращается отрицательное значение. Этот метод может передать исключение `ClassCastException`, если типы сравниваемых объектов не совместимы. Реализуя метод `compare()`, вы можете изменить порядок объектов. Например, чтобы сортировать в обратном порядке, вы можете создать компаратор, который возвращает обратные значения при сравнении.

Метод `equals()`, показанный ниже, проверяет объект на эквивалентность вызывающему компаратору.

```
boolean equals(object объект)
```

Здесь *объект* — это объект, который нужно проверить на эквивалентность. Метод возвращает значение `true`, если *объект* и вызывающий объект представляют собой объекты интерфейса `Comparator` и используют одинаковый способ упорядочения. В противном случае он возвращает значение `false`. Переопределение метода `equals()` не требуется, и большинство простых компараторов в этом не нуждается.

Использование компараторов

Ниже представлен пример, демонстрирующий мощность настраиваемых компараторов. Он реализует метод `compare()` для строк, работающий в порядке, обратном нормальному. То есть он позволяет сортировать элементы набора-дерева в обратном порядке.

```
// Использование настраиваемого компаратора.
import java.util.*;

// Обратный компаратор для строк.
class MyComp implements Comparator<String> {
    public int compare(String a, String b) {
        String aStr, bStr;
        aStr = a;
        bStr = b;
        // Обратное сравнение.
        return bStr.compareTo(aStr);
    }
    // Нет необходимости переопределять equals().
}

class CompDemo {
    public static void main(String args[]) {
        // Создать TreeSet.
        TreeSet<String> ts = new TreeSet<String>(new MyComp());
        // Добавить элементы в набор-дерева.
        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 показывает следующий вывод, дерево теперь отсортировано в обратном порядке.

```
F E D C B A
```

Обратите внимание на класс `MyClass`, который реализует интерфейс `Comparator` и метод `compare()`. (Как упоминалось, переопределение метода `equals()` не является ни необходимым, ни желательным.) Внутри метода `compare()` метод `compareTo()` объекта класса `String` сравнивает две строки. Однако метод `compareTo()` вызывает строка `bStr`, а не строка `aStr`. Это приводит к тому, что результат сравнения получается обратным.

Следующая программа — более полезная, усовершенствованная версия программы с классом `TreeMap`, сохраняющей балансовые счета. В предыдущей версии счета были отсортированы по владельцу счета, но порядок определялся именем. В следующей программе осуществляется сортировка счетов по фамилиям владельцев. Для этого, она использует компаратор, сравнивающий фамилии каждого владельца счета. В результате получается карта, отсортированная по фамилиям.

```
// Использование компаратора для сортировки по фамилиям.
import java.util.*;

// Сравнивает последние два слова в полной строке.
class TComp implements Comparator<String> {
    public int compare(String a, String b) {
        int i, j, k;
        String aStr, bStr;
        aStr = a;
        bStr = b;

        // Найти индекс символа в строке, с которого начинается фамилия.
        i = aStr.lastIndexOf(' ');
        j = bStr.lastIndexOf(' ');
        k = aStr.substring(i).compareTo(bStr.substring(j));
        if(k==0) // Фамилии совпадают, проверить полное имя
            return aStr.compareTo(bStr);
        else
            return k;
    }
    // Нет необходимости переопределять equals().
}

class TreeMapDemo2 {
    public static void main(String args[]) {
        // Создать карту-дерево.
        TreeMap<String, Double> tm = new TreeMap<String,
                                                 Double>(new TComp());

        // Поместить элементы в карту.
        tm.put("Джон Доу", new Double(3434.34));
        tm.put("Том Смит", new Double(123.22));
        tm.put("Джейн Бейкер", new Double(1378.00));
        tm.put("Тод Халл", new Double(99.22));
        tm.put("Ральф Смит", new Double(-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();
```

```
        // Добавить 1000 на счет Джона Доу.
        double balance = tm.get("Джон Доу ");
        tm.put("Джон Доу ", balance + 1000);
        System.out.println("Новый баланс Джона Доу: " +
                           tm.get("John Doe"));
    }
}
```

Вот вывод программы. Обратите внимание на то, что счета теперь отсортированы по фамилиям.

```
Джейн Бейкер: 1378.0
Джон Доу: 3434.34
Ральф Смит: -19.08
Том Смит: 123.22
Тод Халл: 99.22
Новый баланс Джона Доу: 4434.34
```

Компаратор класса `TComp` сравнивает две строки, которые содержат имя и фамилию. Сначала он сравнивает фамилии. Для этого осуществляется поиск позиции последнего пробела в каждой строке с последующим сравнением подстроки, начинающейся с этой позиции. В случае, когда фамилии эквивалентны, сравниваются имена. Это порождает карту-дерево, отсортированную по фамилии, а в пределах одинаковых фамилий — по именам. Вы можете убедиться в этом, поскольку Ральф Смит в выводе программы находится перед Томом Смитом.

Алгоритмы коллекций

Инфраструктура коллекций определяет несколько алгоритмов, которые могут быть применимы к коллекциям и картам. Эти алгоритмы определены как статические методы в классе `Collections`. Они собраны в табл. 17.18. Как упоминалось, начиная с комплекта JDK 5 все алгоритмы были перепроектированы как обобщенные.

Таблица 17.18. Алгоритмы, определенные в классе `Collections`

Метод	Описание
`static <T> boolean addAll(Collection<? super T> c, T... элементы)`	Вставляет элементы, переданные в параметре *элементы*, в коллекцию, указанную в параметре *c*. Возвращает значение `true`, если элементы были добавлены, и значение `false` — в противном случае
`static <T> Queue<T>asLifoQueue(Deque<T> c)`	Возвращает представление коллекции "последний вошел — первый вышел"
`static <T> int binarySearch(List<? extends T> список, T значение, Comparator<? super T> c)`	Ищет в *список* значение *значение* в соответствии с *c*. Возвращает позицию *значение* в *список* или отрицательное значение, если *значение* не найдено
`static <T> int binarySearch(List<? Extends Comparable<? super T>> список, T значение)`	Ищет в *список* значение *значение*. Список должен быть отсортирован. Возвращает позицию *значение* в *список* или отрицательное значение, если *значение* не найдено
`static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> t)`	Возвращает безопасное в отношении типов представление коллекции времени выполнения. Попытка вставить несовместимый элемент вызовет исключение `ClassCastException`

Продолжение табл. 17.18

Метод	Описание
`static <E> List<E> checkedList(List<E> c, Class<E> t)`	Возвращает безопасное в отношении типов представление интерфейса `List` времени выполнения. Попытка вставить несовместимый элемент вызовет исключение `ClassCastException`
`static <K, V> Map<K, V>checkedMap(Map<K, V> c, Class<K> ключT, Class<V> значениеT)`	Возвращает безопасное в отношении типов представление интерфейса `Map` времени выполнения. Попытка вставить несовместимый элемент вызовет исключение `ClassCastException`
`static <E> List<E> checkedSet(Set<E> c, Class<E> t)`	Возвращает безопасное в отношении типов представление интерфейса `Set` времени выполнения. Попытка вставить несовместимый элемент вызовет исключение `ClassCastException`
`static <K, V> SortedMap<K, V>checkedSortedMap(SortedMap<K, V> c, Class<K> ключT, Class<V> значениеT)`	Возвращает безопасное в отношении типов представление интерфейса `SortedMap` времени выполнения. Попытка вставить несовместимый элемент вызовет исключение `ClassCastException`
`static <E> SortedSet<E> checkedSortedSet(SortedSet<E> c, Class<E> t)`	Возвращает безопасное в отношении типов представление интерфейса `SortedSet` времени выполнения. Попытка вставить несовместимый элемент вызовет исключение `ClassCastException`
`static <T> void copy(List<? super T>список1, List<? Extends T> список2)`	Копирует элементы *список2* в *список1*
`static boolean disjoint(Collection<?> a, Collection<?> b)`	Сравнивает элементы в *a* с элементами в *b*. Возвращает значение `true`, если две коллекции не содержат общих элементов (т.е. непересекающиеся множества элементов). В противном случае возвращает значение `false`
`static <T> Enumeration<T> emptyEnumeration()`	Возвращает пустое перечисление, т.е. перечисление без элементов. (Добавлено в JDK 7)
`static <T> Iterator<T> emptyIterator()`	Возвращает пустой итератор, т.е. итератор без элементов. (Добавлено в JDK 7)
`static <T> List<T> emptyList()`	Возвращает неизменяемый, пустой объект интерфейса `List` заданного типа
`static <T> ListIterator<T> emptyListIterator()`	Возвращает пустой итератор списка, т.е. итератор списка без всяких элементов. (Добавлено в JDK 7)
`static <K, V> Map<K, V> emptyMap()`	Возвращает неизменяемый, пустой объект интерфейса `Map` заданного типа
`static <T> Set<T> emptySet()`	Возвращает неизменяемый, пустой объект интерфейса `Set` заданного типа
`static <T> Enumeration<T> enumeration(Collection<T> c)`	Возвращает перечисление на основе *c*. (См. раздел "Интерфейс `Enumeration`" далее в настоящей главе)

Продолжение табл. 17.18

Метод	Описание
`static <T> void fill(List<? super T>`*список*`, T` *объект*`)`	Присваивает *объект* каждому элементу *список*
`static int frequency(Collection<?>` *c*`, object` *объект*`)`	Подсчитывает количество вхождений *объект* в *c* и возвращает результат
`static int indexOfSubList(List<?>`*список*`, List<?>` *вложенныйСписок*`)`	Ищет в *список* первое вхождение *вложенныйСписок*. Возвращает индекс первого совпадения или значение `-1`, если вхождение не найдено
`static int lastIndexOfSubList(List<?>` *список*`, List<?>` *вложенныйСписок*`)`	Ищет в *список* последнее вхождение *вложенныйСписок*. Возвращает индекс первого совпадения или значение `-1`, если вхождение не найдено
`static <T> ArrayList<T>list(Enumeration<T>` *перечисление*`)`	Возвращает объект класса `ArrayList`, содержащий элементы *перечисление*
`static <T> T max(Collection<? extendsT>` *c*`, Comparator<? super T>` *компаратор*`)`	Возвращает максимальный элемент из *c*, определенный при помощи *компаратор*
`static <T extends Object & Comparable<? super T>> Tmax(Collection<? extends T>` *c*`)`	Возвращает максимальный элемент из *c*, определенный естественным порядком. Коллекция должна быть отсортированной
`static <T> T min(Collection<? extends T>` *c*`, Comparator<? super T>` *компаратор*`)`	Возвращает минимальный элемент из *c*, определенный при помощи *компаратор*. Коллекция может быть неотсортированной
`static <T extends Object &Comparable<? superT>>T min(Collection<? extends T>` *c*`)`	Возвращает минимальный элемент из *c*, определенный естественным порядком
`static <T> List<T> nCopies(int` *количество*`, T` *объект*`)`	Возвращает *количество* копий *объект*, содержащихся в неизменяемом списке. Значение параметра *количество* должно быть больше нуля или равно нулю
`static <E> Set<E> newSetFromMap(Map<E, Boolean>` *m*`)`	Создает и возвращает набор на основе *m*, который должен быть пустым на момент вызова метода
`static <T> boolean replaceAll(List<T>`*список*`, T` *старый*`, T` *новый*`)`	Заменяет все вхождения *старый* на *новый* в *список*. Возвращает значение `true`, если выполнена хотя бы одна замена. В противном случае возвращает значение `false`
`static void reverse(List<T>` *список*`)`	Изменяет последовательность элементов в *список* на обратную
`static <T> Comparator<T>reverseOrder(Comparator<T>` *компаратор*`)`	Возвращает компаратор, обратный переданному в *компаратор*. То есть возвращенный компаратор порождает последовательность, обратную той, что делает *компаратор*
`static <T> Comparator<T>reverseOrder()`	Возвращает обратный компаратор, который обращает результат сравнения двух элементов
`static void rotate(List<T>` *список*`, int` *n*`)`	Смещает *список* на *n* позиций вправо. Для смещения влево используйте отрицательное значение *n*

Продолжение табл. 17.18

Метод	Описание
`static void shuffle(List<T> список, Random r)`	Перемешивает (случайным образом) элементы в *список*, используя *r* в качестве источника случайных чисел
`static void shuffle(List<T> список)`	Перемешивает (случайным образом) элементы в *список*
`static <T> Set<T> singleton(T объект)`	Возвращает *объект* как неизменяемый набор. Это простейший способ преобразовать отдельный объект в набор
`static <T> List<T> singletonList(T объект)`	Возвращает *объект* как неизменяемый список. Это простейший способ преобразовать отдельный объект в список
`static <K, V> Map<K, V>singletonMap(K k, V v)`	Возвращает пару "ключ-значение" *k/v* как неизменяемую карту. Это простейший способ преобразовать пару "ключ-значение" в карту
`static <T> void sort(List<T> список, Comparator<? super T> компаратор)`	Сортирует элементы *список* в соответствии с *компаратор*
`static <T extends Comparable<?super T>> void sort(List<T> список)`	Сортирует элементы *список* в соответствии с естественным порядком
`static void swap(List<T> список, int индекс1, int индекс2)`	Меняет местами элементы *список*, находящиеся в позициях *индекс1* и *индекс2*
`static <T> Collection<T> synchronizedCollection(Collection<T> c)`	Возвращает безопасную в отношении потоков коллекцию, наполненную элементами *c*
`static <T> List<T> synchronizedList(List<T> список)`	Возвращает безопасный в отношении потоков список, наполненный элементами *список*
`static <K, V> Map<K, V>synchronizedMap(Map<K, V> m)`	Возвращает безопасную в отношении потоков карту, наполненную элементами *m*
`static <T> Set<T> synchronizedSet(Set<T> s)`	Возвращает безопасный в отношении потоков набор, наполненный элементами *s*
`static <K, V> SortedMap<K, V>synchronizedSortedMap(SortedMap<K, V> sm)`	Возвращает безопасную в отношении потоков *отсортированную карту, наполненную элементами sm*
`static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> ss)`	Возвращает безопасный в отношении потоков отсортированный набор, наполненный элементами *ss*
`static <T> Collection<T> unmodifiable Collection(Collection<? extends T> c)`	Возвращает немодифицируемую коллекцию, наполненную элементами *c*
`static <T> List<T> unmodifiableList(List<? extends T> список)`	Возвращает немодифицируемый список, наполненный элементами *список*
`static <K, V> Map<K, V>unmodifiableMap(Map<? extends K, ?extends V> m)`	Возвращает немодифицируемую карту, наполненную элементами *m*
`static <T> Set<T> unmodifiableSet(Set<? extends T> s)`	Возвращает немодифицируемый набор, наполненный элементами *s*

Метод	Описание
`static <K, V> SortedMap<K, V>unmodifiableSortedMap(SortedMap<K, ?extends V> sm)`	Возвращает немодифицируемую отсортированную карту, наполненную элементами *sm*
`static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> ss)`	Возвращает немодифицируемый отсортированный набор, наполненный элементами *ss*

При попытке сравнения несовместимых типов некоторые из этих методов могут передать исключение `ClassCastException` либо исключение `UnsupportedOperationException`, при попытке модифицировать немодифицируемые коллекции. В зависимости от метода, возможны и другие исключения.

Особое внимание следует уделить набору перегруженных методов, таких как метод `checkedCollection()`, который возвращает то, что в документации по API именуется "динамическим представлением, безопасным в отношении типов" коллекций. Это представление является ссылкой на коллекцию, которая во время выполнения отслеживает вставку объектов в коллекцию на предмет совместимости типов. Попытка вставить несовместимый элемент вызовет исключение `ClassCastException`. Использование этого представления полезно при отладке, так как гарантирует, что коллекция всегда содержит корректные элементы. К этим методам относятся `checkedSet()`, `checkedList()`, `checkedMap()` и т.д. Они позволяют получить безопасное в отношении типов представление указанной коллекции.

Отметим, что некоторые методы, такие как `synchronizedList()` и `synchronizedSet()`, служат для получения синхронизированных (безопасных в отношении потоков) копий различных коллекций. Как упоминалось, стандартные реализации коллекций, как правило, не синхронизированы. Для обеспечения синхронизации следует применять синхронизирующий алгоритм. И еще один момент: итераторы для синхронизированных коллекций должны использоваться в пределах блоков `synchronized`.

Набор методов, начинающихся с `unmodifiable`, возвращает представления различных коллекций, которые не могут быть модифицированы. Это может оказаться удобным, если вы хотите гарантировать доступ к коллекциям только для чтения, без права записи.

В интерфейсе `Collection` определены три статические переменные: `EMPTY_SET`, `EMPTY_LIST` и `EMPTY_MAP`. Все они являются константами.

В следующей программе демонстрируются некоторые алгоритмы. Программа создает и инициирует связный список. Метод `reverseOrder()` возвращает объект интерфейса `Comparator`, который обращает сравнение объектов класса `Integer`. Элементы списка сначала сортируются согласно этому компаратору, а затем отображаются. Далее этот список тасуется вызовом метода `shuffle()`, после чего отображаются его минимальное и максимальное значения.

```
// Демонстрация применения различных алгоритмов.
import java.util.*;

class AlgorithmsDemo {
    public static void main(String args[]) {
        // Создать неинициализированный связный список.
        LinkedList<Integer> ll = new LinkedList<Integer>();
        ll.add(-8);
        ll.add(20);
        ll.add(-20);
        ll.add(8);
```

```
        // Создать компаратор обратного порядка.
        Comparator<Integer> r = Collections.reverseOrder();

        // Сортировать список этим компаратором.
        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 перемешанный список.
        System.out.print("Список перемешан: ");

        for(int i : ll)
            System.out.print(i + " ");

        System.out.println();
        System.out.println("Минимум: " + Collections.min(ll));
        System.out.println("Максимум: " + Collections.max(ll));
    }
}
```

Ниже приведен вывод этой программы.

```
Список отсортирован в обратном порядке: 20 8 -8 -20
Список перемешан: 20 -20 8 -8
Минимум: -20
Максимум: 20
```

Обратите внимание на то, что методы `min()` и `max()` оперируют списком после того, как он был перемешан. Ни один из этих методов не требует, чтобы список был отсортирован.

Класс `Arrays`

Этот класс предоставляет разнообразные удобные методы для работы с массивами. Эти методы помогают заполнить пробел между коллекциями и массивами. Каждый метод, определенный в классе `Arrays`, рассматривается далее в разделе.

Метод `asList()` возвращает список, наполненный элементами указанного массива. Другими словами, и список, и массив ссылаются на одно и то же. Он имеет следующую сигнатуру.

```
static <T> List asList(T... массив)
```

Здесь *массив* — это массив, содержащий данные.

Метод `binarySearch()` использует бинарный поиск для нахождения заданного значения. Этот метод должен применяться к отсортированным массивам. Он имеет следующие формы (дополнительные формы обеспечивают поиск в диапазоне значений).

```
static int binarySearch(byte массив[ ], byte значение)
static int binarySearch(char массив[ ], char значение)
static int binarySearch(double массив[ ], double значение)
static int binarySearch(float массив[ ], float значение)
static int binarySearch(int массив[ ], int значение)
```

```
static int binarySearch(long массив[ ], long значение)
static int binarySearch(short массив[ ], short значение)
static int binarySearch(Object массив[ ], Object значение)
static <T> int binarySearch(T[ ] массив, T значение, Comparator<? super T> с)
```

Здесь *массив* — это массив, в котором осуществляется поиск, а *значение* — значение, которое нужно найти. Последние две формы передают исключение `ClassCastException`, если *массив* содержит элементы, которые невозможно сравнивать (например, классов `Double` и `StringBuffer`), либо *значение* несовместимо с типами *массив*. В последней форме интерфейс `Comparator` *с* используется для определения порядка элементов в массиве *массив*. Во всех классах, если *значение* содержится в *массив*, возвращается индекс элемента. В противном случае возвращается отрицательное значение.

Метод `copyOf()` возвращает копию массива и имеет следующие формы.

```
static boolean[ ] copyOf(boolean[ ] источник, int длина)
static byte[ ] copyOf(byte[ ] источник, int длина)
static char[ ] copyOf(char[ ] источник, int длина)
static double[ ] copyOf(double[ ] источник, int длина)
static float[ ] copyOf(float[ ] источник, int длина)
static int[ ] copyOf(int[ ] источник, int длина)
static long[ ] copyOf(long[ ] источник, int длина)
static short[ ] copyOf(short[ ] источник, int длина)
static <T> T[ ] copyOf(T[ ] источник, int длина)
static <T, U> T[ ] copyOf(U[ ] источник, int длина, Class<? extends T[ ]>
типРезульт)
```

Исходный массив задан параметром *источник*, а длина копии — в параметре *длина*. Если копия длиннее, чем *источник*, она дополняется нулями (для числовых массивов), значениями `null` (для массивов объектов) или значениями `false` (для булевых массивов). Если копия короче, чем *источник*, она усекается. В последней форме тип *типРезульт* становится типом возвращаемого массива. Если *длина* отрицательна, передается исключение `NegativeArraySizeException`. Если *источник* содержит значение `null`, передается исключение `NullPointerException`. Если тип *типРезульт* не совместим с типом *источник*, передается исключение `ArrayStoreException`.

Метод `copyOfRange()` также возвращает копию диапазона внутри массива и имеет следующие формы.

```
static boolean[ ] copyOfRange(boolean[ ] источник, int начало, int конец)
static byte[ ] copyOfRange(byte[ ] источник, int начало, int конец)
static char[ ] copyOfRange(char[ ] источник, int начало, int конец)
static double[ ] copyOfRange(double[ ] источник, int начало, int конец)
static float[ ] copyOfRange(float[ ] источник, int начало, int конец)
static int[ ] copyOfRange(int[ ] источник, int начало, int конец)
static long[ ] copyOfRange(long[ ] источник, int начало, int конец)
static short[ ] copyOfRange(short[ ] источник, int начало, int конец)
static <T> T[ ] copyOfRange(T[ ] источник, int начало, int конец)
static <T, U> T[ ] copyOfRange(U[ ] источник, int начало, int конец,
Class<? extends T[ ]> типРезульт)
```

Исходный массив задан параметром *источник*. Диапазон для копирования задан индексами, передаваемыми параметрами *начало* и *конец*. Диапазон распространяется от *начало* до *конец*-1. Если диапазон длиннее, чем *источник*, копия дополняется нулями (для числовых массивов), значениями `null` (для массивов объектов) или значениями `false` (для булевых массивов). В последней форме тип *типРезульт* становится типом возвращаемого массива.

Если *начало* отрицательно или больше длины *источник*, передается исключение `ArrayIndexOutOfBoundsException`.

Если *начало* больше *конец*, передается исключение IllegalArgumentException. Если *источник* содержит значение null, передается исключение NullPointerException. Если *типРезульт* не совместим с типом *источник*, передается исключение ArrayStoreException.

Метод equals() возвращает значение true, если два массива эквивалентны. В противном случае он возвращает значение false.

Различные формы метода equals() показаны ниже.

```
static boolean equals(boolean массив1[], boolean массив2[])
static boolean equals(byte массив1[], byte массив2[])
static boolean equals(char массив1[], char массив2[])
static boolean equals(double массив1[], double массив2[])
static boolean equals(float массив1[], float массив2[])
static boolean equals(int массив1[], int массив2[])
static boolean equals(long массив1[], long массив2[])
static boolean equals(short массив1[], short массив2[])
static boolean equals(Object массив1[], Object массив2[])
Здесь массив1 и массив2 – массивы, сравниваемые на эквивалентность.
```

Метод deepEquals() может быть использован для определения того, являются ли два массива, которые могут содержать вложенные массивы, эквивалентными. Он имеет следующее объявление.

```
static boolean deepEquals(Object[] a, Object[] b)
```

Метод возвращает значение true, если переданные ему массивы *a* и *b* эквивалентны. Если массивы *a* и *b* содержат вложенные массивы, они также сравниваются. Если массивы *a* и *b* либо их вложенные массивы отличаются, метод возвращает значение false.

Метод fill() присваивает значение всем элементам массива. Другими словами, он заполняет массив указанным значением. Метод fill() имеет две версии. Первая версия, формы которой представлены ниже, заполняет весь массив.

```
static void fill(boolean массив[], boolean значение)
static void fill(byte массив[], byte значение)
static void fill(char массив[], char значение)
static void fill(double массив[], double значение)
static void fill(float массив[], float значение)
static void fill(int массив[], int значение)
static void fill(long массив[], long значение)
static void fill(short массив[], short значение)
static void fill(Object массив[], Object значение)
```

Здесь *значение* присваивается всем элементам *массив*.

Вторая версия метода fill() присваивает значение подмножеству массива. Его формы перечислены ниже.

```
static void fill(boolean массив[], int начало, int конец, boolean значение)
static void fill(byte массив[], int начало, int конец, byte значение)
static void fill(char массив[], int начало, int конец, char значение)
static void fill(double массив[], int начало, int конец, double значение)
static void fill(float массив[], int начало, int конец, float значение)
static void fill(int массив[], int начало, int конец, int значение)
static void fill(long массив[], int начало, int конец, long значение)
static void fill(short массив[], int начало, int конец, short значение)
static void fill(Object массив[], int начало, int конец, Object значение)
```

Здесь *значение* присваивается элементам *массив* от позиции *начало* до *конец*-1. Все эти методы могут передать исключение IllegalArgumentException, если *начало* больше *конец*, либо ArrayIndexOutOfBoundsException, если *начало* или *конец* выходят за пределы массива. Исключение ArrayStoreExceptionis возможно и с версиями класса Object.

Метод `sort()` сортирует массив таким образом, что он упорядочивается в возрастающем порядке. Метод `sort()` имеет две версии. Первая версия, показанная ниже, сортирует весь массив.

```
static void sort(byte массив[])
static void sort(char массив[])
static void sort(double массив[])
static void sort(float массив[])
static void sort(int массив[])
static void sort(long массив[])
static void sort(short массив[])
static void sort(Object массив[])
static <T> void sort(T массив[], Comparator<? super T> с)
```

Здесь *массив* — это массив, подлежащий сортировке. В последней форме *с* — это интерфейс `Comparator`, который используется для упорядочения элементов *массив*. Последние две формы могут передавать исключение `ClassCastException`, если элементы сортируемого массива несовместимы.

Вторая версия метода `sort()` позволяет указать диапазон массива, который вы хотите сортировать. Его формы представлены ниже.

```
static void sort(byte массив[], int начало, int конец)
static void sort(char массив[], int начало, int конец)
static void sort(double массив[], int начало, int конец)
static void sort(float массив[], int начало, int конец)
static void sort(int массив[], int начало, int конец)
static void sort(long массив[], int начало, int конец)
static void sort(short массив[], int начало, int конец)
static void sort(Object массив[], int начало, int конец)
static <T> void sort(T массив[], int начало, int конец, Comparator<?
super T> c)
```

Здесь будет отсортирован диапазон элементов массива, начинающийся с *начало* и заканчивающийся *конец*-1. В последней форме *с* — это интерфейс `Comparator`, который используется для определения порядка элементов массива.

Все эти методы могут передавать исключение `IllegalArgumentException`, если *начало* больше *конец*, либо `ArrayIndexOutOfBoundsException`, если *начало* или *конец* выходят за пределы массива. Последние две формы также могут передать исключение `ClassCastException`, если сортируемые элементы массива не совместимы.

Класс `Arrays` предоставляет также методы `toString()` и `hashCode()` для различных типов массивов. Кроме того, в нем предусмотрены методы `deepToString()` и `deepHashCode()`, которые эффективно работают с массивами, имеющими вложенные массивы.

В следующей программе иллюстрируется применение некоторых методов класса `Arrays`.

```
// Демонстрация применения Arrays.
import java.util.*;

class ArraysDemo {
    public static void main(String args[]) {
        // Распределить и инициировать массив.
        int array[] = new int[10];
        for(int i = 0; i < 10; i++)
            array[i] = -3 * i;

        // Отобразить, отсортировать и вновь отобразить массив.
        System.out.print("Исходное содержимое: ");
        display(array);
```

```
        Arrays.sort(array);
        System.out.print("Отсортированный массив: ");
        display(array);

        // Наполнение и отображение массива.
        Arrays.fill(array, 2, 6, -1);
        System.out.print("После fill(): ");
        display(array);

        // Сортировать и отобразить массив.
        Arrays.sort(array);
        System.out.print("После повторной сортировки: ");
        display(array);

        // Бинарный 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 этой программы.

```
Исходное содержимое: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Отсортированный массив: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
После fill(): -27 -24 -1 -1 -1 -1 -9 -6 -3 0
После повторной сортировки: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
Значение -9 находится в позиции 2
```

Зачем нужны обобщенные коллекции

Как уже упоминалось в начале этой главы, в комплекте JDK 5 инфраструктура коллекций была перепроектирована для учета обобщений. Более того, инфраструктура коллекций — возможно, единственное наиболее важное применение обобщений в Java API. Причина этого в том, что обобщения обеспечивают безопасность типов инфраструктуры коллекций. Прежде чем двигаться дальше, стоит потратить некоторое время на детальное рассмотрение сущности этого усовершенствования. Это также объясняет, почему прежний код, написанный до появления обобщенных коллекций, должен быть модифицирован.

Давайте начнем с примера, который использует старый код, до введения обобщений. В следующей программе список строк сохраняется в объекте класса `ArrayList`, а затем отображается содержимое списка.

```
// Пример использования коллекций до введения обобщений.
import java.util.*;
class OldStyle {
    public static void main(String args[]) {
        ArrayList list = new ArrayList();
        // lines хранит строки, но могут быть сохранены объекты любого
        // типа. В коде старого стиля нет возможности защитить тип
        // сохраняемых в коллекции объектов.
        list.add("один");
        list.add("два");
```

```
        list.add("три");
        list.add("четыре");
        Iterator itr = list.iterator();

        while(itr.hasNext()) {
            // Чтобы извлечь элемент, требуется явное приведение типов,
            // потому что коллекции хранят только Object.
            String str = (String) itr.next(); // здесь необходимо явное
                                              //приведение.
            System.out.println(str + " имеет длину " + str.length() +
                               " символов.");
        }
    }
}
```

До введения обобщенного синтаксиса все коллекции хранили ссылки на объекты класса `Object`. Это позволяло сохранять в них ссылки на объекты любого типа. Приведенная выше программа использует это свойство для хранения в списке объектов класса `String`, но также может сохранять ссылки любого другого типа.

К сожалению, тот факт, что коллекции старого вида сохраняют только объекты класса `Object`, может легко приводить к ошибкам. В первую очередь, это требует, чтобы вы, а не компилятор, обеспечивали сохранение в конкретной коллекции только объектов правильных типов. Например, в предыдущем примере список однозначно предназначен для сохранения объектов класса `String`, но нет ничего, что действительно предотвратило бы добавление в коллекцию ссылок на объекты других типов. Так, например, компилятор не обнаружит ничего недопустимого в следующей строке кода.

```
list.add(new Integer(100));
```

Поскольку объект `list` сохраняет ссылки на класс `Object`, он может сохранить ссылку на класс `Integer` — точно так же, как и ссылку на класс `String`. Но если вы предполагаете хранить в списке только строки, то предыдущий оператор повредит вашу коллекцию. К тому же компилятор не имеет никакой возможности узнать, что этот оператор неверен.

Вторая проблема со старыми (до введения обобщений) коллекциями в том, что когда вы получаете ссылку из коллекции, то должны вручную явно приводить эту ссылку к правильному типу. Вот почему предыдущая программа приводит ссылку, полученную от метода `next()`, к типу `String`. До введения обобщений коллекции просто сохраняли ссылки на класс `Object`. Поэтому при извлечении объектов из коллекций необходимо было приведение типа.

Помимо неудобств, связанных с необходимостью приведения типов ссылок, этот недостаток информации о типе объектов приводит к довольно серьезным и неожиданно легко порождаемым ошибкам. Поскольку класс `Object` может быть приведен к любому объектному типу, всегда существовала возможность привести полученную из коллекции ссылку к *неверному типу*. Например, если к предыдущему примеру добавить следующий оператор, пример будет по-прежнему компилироваться без ошибок, но передавать исключение времени выполнения во время запуска программы.

```
Integer i = (Integer) itr.next();
```

Вспомните, что в предыдущем примере в объекте `list` сохранялись только ссылки на экземпляры класса `String`. Поэтому при попытке приведения типа `String` к типу `Integer` передается исключение неверного приведения типов. Так как это случится во время выполнения программы, подобная ошибка весьма серьезна.

Добавление обобщений кардинально повышает удобство использования и безопасность коллекций.

- Гарантирует, что в коллекции будут сохранены только ссылки на объекты правильного типа. То есть коллекции всегда будут содержать ссылки известного типа.
- Исключает необходимость приведения типов ссылок, извлеченных из коллекции. Вместо этого ссылка, извлеченная из коллекции, будет автоматически приведена к правильному типу. Это предотвращает ошибки времени выполнения по причине неверного приведения и позволяет избежать целого ряда подобных ошибок.

Эти два усовершенствования стали возможны благодаря тому, что каждому классу коллекций может быть передан параметр, задающий тип коллекции. Например, класс `ArrayList` теперь объявляется следующим образом.

```
class ArrayList<E>
```

Здесь *E* — тип элемента, сохраняемого в коллекции. Таким образом, следующая строка объявляет объект класса `ArrayList` для объектов класса `String`.

```
ArrayList<String> list = new ArrayList<String>();
```

Теперь в список могут быть добавлены только ссылки на объекты класса `String`.

Интерфейсы `Iterator` и `ListIterator` теперь также стали обобщенными. Это означает, что параметр типа должен быть согласован с типом коллекции, для которой получен итератор. Более того, эта совместимость типов принудительно обеспечивается во время компиляции.

Следующая программа показывает современную, обобщенную, форму предыдущей программы.

```
// Современная, обобщенная, версия.
import java.util.*;

class NewStyle {
    public static void main(String args[]) {
        // Теперь list содержит ссылки типа String.
        ArrayList<String> list = new ArrayList<String>();
        list.add("один");
        list.add("два");
        list.add("три");
        list.add("четыре");

        // Отметим, что Iterator также обобщенный.
        Iterator<String> itr = list.iterator();

        // Следующий оператор теперь вызовет ошибку времени компиляции.
        // Iterator<Integer> itr = list.iterator(); // Ошибка!
        while(itr.hasNext()) {
            String str = itr.next(); // приведение не требуется
            // Теперь следующая строка породит ошибку компиляции, а не
            //времени выполнения

            // Integer i = itr.next(); // Это не откомпилируется
            System.out.println(str + " имеет длину " + str.length() +
                               " символов.");
        }
    }
}
```

Теперь список может содержать только ссылки на объекты класса `String`. Более того, как видно из следующей строки, нет необходимости приводить тип объекта, возвращаемого методом `next()`, к типу `String`.

```
String str = itr.next(); //приведение не нужно
```

Приведение выполняется автоматически.

Поскольку существует поддержка базовых типов, прежний код, написанный до появления обобщенных коллекций, продолжает компилироваться и выполняться. Однако весь новый код должен быть обобщенным, а старый вы можете обновлять постепенно, при наличии свободного времени. Добавление обобщений в инфраструктуру коллекций — фундаментальное усовершенствование, которое следует использовать везде, где это возможно.

Унаследованные классы и интерфейсы

Как уже объяснялось в начале главы, ранние версии пакета `java.util` не включали в себя инфраструктуру коллекций. Вместо нее было определено несколько классов и интерфейсов, обеспечивающих специальные методы хранения объектов. Когда были добавлены коллекции (начиная с J2SE 1.2), некоторые исходные классы были перепроектированы для поддержки интерфейсов коллекций. То есть сейчас они, технически, являются частью инфраструктуры Collections Framework. Тем не менее, хотя новые коллекции дублируют функции устаревших классов, обычно вы будете использовать классы новых коллекций. Вообще, унаследованные классы поддерживаются потому, что существует код, использующий их.

Еще один момент: ни один из современных классов коллекции, описанных в этой главе, не синхронизируется, а все устаревшие классы синхронизируются. Это различие может быть важным в некоторых ситуациях. Конечно, вы можете легко синхронизировать коллекции, используя один из алгоритмов, представленных интерфейсом `Collection`.

Устаревшие классы, определенные в пакете `java.util`, показаны ниже.

`Dictionary`	`Hashtable`	`Properties`	`Stack`	`Vector`

Есть также один унаследованный интерфейс, называемый `Enumeration`. Следующие разделы рассматривают этот интерфейс и каждый из унаследованных классов по очереди.

Интерфейс `Enumeration`

Рассматриваемый интерфейс определяет методы, которыми вы можете перебирать (получая по одному за раз) элементы в коллекции объектов. Этот унаследованный интерфейс был замещен интерфейсом `Iterator`. Хотя интерфейс `Enumeration` и применяется, но считается устаревшим для нового кода. Однако он используется несколькими методами унаследованных классов (таких, как класс `Vector` или `Properties`), а также некоторыми другими классами API и широко используется существующим кодом приложений. Поскольку этот интерфейс все еще задействован, он был перепроектирован в обобщенном виде для комплекта JDK 5. Интерфейс `Enumeration` имеет следующее объявление.

```
interface Enumeration<E>
```

Здесь *E* определяет тип элементов, которые будут перебираться.

В интерфейсе `Enumeration` определены следующие два метода.

```
boolean hasMoreElements()
E nextElement()
```

При реализации метод `hasMoreElements()` должен возвращать значение `true` до тех пор, пока остаются элементы, подлежащие извлечению, и значение `false` — когда все элементы уже перечислены. Метод `nextElement()` возвращает следующий объект перечисления. То есть каждый вызов метода `nextElement()` получает следующий объект перечисления. По завершении перебора перечисления этот метод передает исключение `NoSuchElementException`.

Класс `Vector`

Этот класс реализует динамический массив. Он подобен классу `ArrayList`, но имеет два отличия: класс `Vector` синхронизирован и включает много устаревших методов, которые дублируют функции методов, определенных инфраструктурой Collections Framework. С появлением коллекций класс `Vector` был перепроектирован как расширение класса `AbstractList`, и в него была добавлена реализация интерфейса `List`. В версии JDK 5 он был перепроектирован под применение обобщенного синтаксиса, и в нем появилась реализация интерфейса `Iterable`. Это означает, что класс `Vector` стал полностью совместимым с коллекциями и может выдавать свое содержимое в усовершенствованном цикле `for`.

Класс `Vector` объявлен следующим образом.

```
class Vector<E>
```

Здесь *E* определяет тип элементов, которые будут храниться.

Ниже перечислены конструкторы класса `Vector`.

```
Vector()
Vector(int размер)
Vector(int размер, int инкремент)
Vector(Collection<? extends E> c)
```

Первая форма создает вектор по умолчанию, имеющий начальный размер 10. Вторая форма создает вектор, начальная емкость которого равна *размер*. Третья форма создает вектор с начальной емкостью *размер*, а инкремент указан в параметре *инкремент*. Инкремент задает количество элементов, которые будут резервироваться при каждом увеличении размера вектора. Четвертая форма создает вектор, содержащий элементы коллекции *c*.

Все векторы начинаются с некоторой начальной емкости. Когда эта емкость заполнена, при следующей попытке сохранения объекта в векторе он автоматически увеличивает количество выделенного пространства по мере роста вектора. Это увеличение важно, поскольку резервирование памяти — дорогостоящая по времени операция. Общий объем дополнительного пространства памяти, выделенного при каждом резервировании, задается величиной инкремента, указанного при создании вектора. Если вы не указываете размер инкремента, размер вектора удваивается при каждом цикле резервирования памяти.

В классе `Vector` определены следующие защищенные переменные члены.

```
int capacityIncrement;
int elementCount;
Object[] elementData;
```

Значение инкремента сохраняется в переменной `capacityIncrement`. Количество элементов, находящихся в данный момент в векторе, хранится в переменной `elementCount`. Массив, хранящий сам вектор, содержится в переменной `elementData`.

В дополнение к методам коллекций, определенных в интерфейсе `List`, класс `Vector` определяет несколько унаследованных методов, которые перечислены в табл. 17.19.

Поскольку класс `Vector` реализует интерфейс `List`, вы можете использовать его так же, как применяете экземпляр класса `ArrayList`. Можно также манипулировать им, используя один из унаследованных методов. Например, после создания экземпляра класса `Vector` вы можете добавлять элементы к нему с помощью метода `addElement()`. Чтобы получить элемент, расположенный в определенной позиции, вызовите метод `elementAt()`. Чтобы получить первый элемент вектора, примените метод `firstElement()`, а чтобы последний – метод `lastElement()`. Индекс определенного элемента можно получить методом `indexOf()` или `lastIndexOf()`. Чтобы удалить элемент вектора, вызовите метод `removeElement()` или `removeElementAt()`.

Таблица 17.19. Унаследованные методы, определенные в классе `Vector`

Метод	Описание
`void addElement(E` *элемент*`)`	Объект, указанный в *элемент*, добавляется к вектору
`int capacity()`	Возвращает емкость вектора
`Object clone()`	Возвращает дубликат вызывающего объекта
`boolean contains(Object` *элемент*`)`	Возвращает значение `true`, если *элемент* содержится в векторе, и значение `false` – в противном случае
`void copyInto(Object` *массив*`[])`	Элементы, содержащиеся в вызывающем векторе, копируются в массив *массив*
`E elementAt(int` *индекс*`)`	Возвращает элемент, расположенный в позиции *индекс*
`Enumeration<E> elements()`	Возвращает перечисление элементов вектора
`void ensureCapacity(int` *размер*`)`	Устанавливает минимальную емкость вектора равной *размер*
`E firstElement()`	Возвращает первый элемент вектора
`int indexOf(Object` *элемент*`)`	Возвращает индекс первого вхождения *элемент*. Если объект не найден в векторе, возвращает значение `-1`
`int indexOf(Object` *элемент*`, int` *начало*`)`	Возвращает индекс первого вхождения *элемент* после *начало*. Если объект не найден в векторе, возвращает значение `-1`
`void insertElementAt(E` *элемент*`, int` *индекс*`)`	Добавляет *элемент* к вектору в позицию *индекс*
`boolean isEmpty()`	Возвращает значение `true`, если вектор пуст, и значение `false` – в противном случае
`E lastElement()`	Возвращает последний элемент вектора
`int lastIndexOf(Object` *элемент*`)`	Возвращает индекс последнего вхождения *элемент*. Если объект не найден в векторе, возвращает значение `-1`
`int lastIndexOf(Object` *элемент*`, int` *начало*`)`	Возвращает индекс последнего вхождения *элемент* перед *начало*. Если объект не найден в векторе, возвращает значение `-1`
`void removeAllElements()`	Очищает вектор. После выполнения этого метода размер вектора равен нулю
`boolean removeElement(Object` *элемент*`)`	Удаляет *элемент* из вектора. Если в векторе содержится более одного экземпляра данного элемента, то удаляется только первый из них. Возвращает значение `true`, если элемент удален, и значение `false` – если объект не найден

Окончание табл. 17.19

Метод	Описание
`void removeElementAt(int` *индекс*`)`	Удаляет из вектора *элемент*, расположенный в позиции *индекс*
`void setElementAt(E` *элемент*`, int` *индекс*`)`	Устанавливается значение элемента в позиции *индекс*
`void setSize(int` *размер*`)`	Устанавливается количество элементов вектора в *размер*. Если новый размер меньше старого, элементы теряются. Если же новый размер больше старого, добавляются пустые элементы
`int size()`	Возвращает количество элементов, содержащихся в векторе
`String toString()`	Возвращает строковый эквивалент значения вектора
`void trimToSize()`	Устанавливает емкость вектора равной количеству элементов, содержащихся в нем на данный момент

В следующей программе вектор используется для сохранения разных типов числовых объектов. В ней демонстрируется несколько унаследованных методов, определенных в классе `Vector`. Кроме того, в программе показана работа с интерфейсом `Enumeration`.

```
// Демонстрация различных операций с Vector.
import java.util.*;

class VectorDemo {
    public static void main(String args[]) {

        // Начальный размер 3, инкремент 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 3.");

        // Перебор элементов вектора.
        Enumeration<Integer> vEnum = v.elements();

        System.out.println("\nЭлементы вектора:");
        while(vEnum.hasMoreElements())
            System.out.print(vEnum.nextElement() + " ");
        System.out.println();
    }
}
```

Вывод этой программы приведен ниже.

```
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 3.
Элементы вектора:
1 2 3 4 5 6 7 9 10 11 12
```

Вместо того чтобы полагаться на перебор объектов в цикле (как это делает приведенная выше программа), вы можете использовать итератор. Например, в программу можно поместить следующий итеративный код.

```
// Использовать итератор для отображения содержимого
Iterator<Integer> vItr = v.iterator();

System.out.println("\nЭлементы вектора:");
while(vItr.hasNext())
    System.out.print(vItr.next() + " ");
System.out.println();
```

Вы можете перебрать вектор циклом "for-each", как показано в следующей версии предыдущего кода.

```
// Использовать расширение цикла for для отображения элементов
System.out.println("\nЭлементы вектора:");
for(int i : v)
    System.out.print(i + " ");

System.out.println();
```

Поскольку интерфейс `Enumeration` не рекомендуется применять в новом коде, обычно вы будете использовать итераторы и циклы "for-each" для перебора всех элементов вектора. Конечно, существует большой объем кода, использующего интерфейс `Enumeration`. К счастью, перечисления и итераторы работают почти одинаково.

Класс `Stack`

Класс `Stack` — это подкласс класса `Vector`, который реализует стандартный стек "последний вошел — первый вышел". Класс `Stack` определяет только стандартный конструктор, создающий пустой стек. С появлением версии JDK 5 класс

Stack был перепроектирован под обобщенный синтаксис, и теперь он объявлен следующим образом.

```
class Stack<E>
```

Здесь *E* указывает тип элементов, сохраняемых в стеке. Класс Stack включает все методы, определенные в классе Vector, и добавляет некоторые свои, которые перечислены в табл. 17.20.

Таблица 17.20. Методы, определенные в классе Stack

Метод	Описание
boolean empty()	Возвращает значение true, если стек пустой, и значение false, если он содержит элементы
E peek()	Возвращает элемент с вершины стека, но не удаляет его
E pop()	Возвращает элемент с вершины стека, удаляя его
E push(E *элемент*)	Вталкивает *элемент* в стек. Значение параметра *элемент* также возвращается
int search(Object *элемент*)	Ищет *элемент* в стеке. Если *элемент* найден, возвращает смещение от вершины стека до этого элемента. В противном случае возвращает значение -1

Чтобы поместить объект в верхушку стека, вызовите метод push(). Чтобы удалить и вернуть верхний элемент, вызовите метод pop(). Вы можете использовать метод peek() для возврата верхнего объекта без его удаления. Исключение EmptyStackException передается, если применить метод pop() или peek() к пустому стеку. Метод empty() возвращает значение true, если стек пуст. Метод search() определяет, содержится ли объект в стеке, и возвращает количество вызовов метода pop(), необходимых для перемещения его в вершину стека. Ниже приведен пример, который создает стек, вставляет в него несколько объектов класса Integer, а затем извлекает их обратно.

```
// Демонстрация применения 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 представлен вывод программы: обратите внимание на вызов обработчика исключения `EmptyStackException`, позволяющего успешно обработать ситуацию с пустым стеком.

```
стек: []
push(42)
стек: [42]
push(66)
стек: [42, 66]
push(99)
стек: [42, 66, 99]
BookNew_pop -> 99
стек: [42, 66]
pop -> 66
стек: [42]
pop -> 42
стек: []
pop -> стек пуст
```

Отметим еще один момент: хотя класс `Stack` в версии Java SE 6 устаревшим не считается, лучшим выбором будет класс `ArrayDeque`.

Класс `Dictionary`

Класс `Dictionary` — это абстрактный класс, представляющий хранилище для пар "ключ-значение" и работающий, в основном, подобно карте. Передав ключ и значение, вы можете сохранить значение в объекте класса `Dictionary`. Однажды сохраненное значение можно извлечь по его ключу. То есть, подобно карте, объект класса `Dictionary` (словарь) можно считать списком пар "ключ-значение". Хотя класс `Dictionary` и не объявлен нежелательным, его можно рассматривать как устаревший, поскольку его полностью заменяет карта. Однако класс `Dictionary` все еще применяется, поэтому мы опишем его здесь.

С появлением комплекта JDK 5 класс `Dictionary` был также сделан обобщенным. Он объявлен следующим образом.

```
class Dictionary<K, V>
```

Здесь *K* указывает тип ключей, а *V* — тип значений. Абстрактные методы, определенные в классе `Dictionary`, перечислены в табл. 17.21.

Таблица 17.21. Методы, определенные в классе `Dictionary`

Метод	Описание
`Enumeration<V> elements()`	Возвращает перечисление значений, хранимых в словаре
`V get(Object` *ключ*`)`	Возвращает объект, содержащий значение, ассоциированное с *ключ*. Если *ключ* не содержится в словаре, возвращается значение `null`

Окончание табл. 17.21

Метод	Описание
`boolean isEmpty()`	Возвращает значение `true`, если словарь пуст, и значение `false`, если он содержит хотя бы одно значение
`Enumeration<K> keys()`	Возвращает перечисление ключей, хранимых в словаре
`V put(K` *ключ*`, V` *значение*`)`	Вставляет ключ и значение в словарь. Возвращает значение `null`, если *ключ* еще не содержится в словаре, в противном случае возвращает предыдущее значение, ассоциированное с *ключ*
`V remove(Object` *ключ*`)`	Удаляет *ключ* и его значение из словаря. Возвращает значение, ассоциированное с *ключ*. Если *ключ* не содержится в словаре, возвращает значение `null`
`int size()`	Возвращает количество элементов в словаре

Чтобы добавить ключ и его значение в словарь, используйте метод `put()`. Вызывайте метод `get()` для извлечения значения по заданному ключу. Ключи и значения могут быть возвращены как перечисление методами `keys()` и `elements()` соответственно. Метод `size()` возвращает количество пар "ключ-значение", сохраненных в словаре, а метод `isEmpty()` — значение `true`, если словарь пуст. Для удаления любой пары "ключ-значение" можно применять метод `remove()`.

Помните! Класс `Dictionary` устарел. Для получения функциональных возможностей хранения пар "ключ-значение" следует реализовать интерфейс `Map`.

Класс `Hashtable`

Класс `Hashtable` — это часть исходного пакета `java.util` и конкретная реализация класса `Dictionary`. Однако с появлением коллекций класс `Hashtable` был перепроектирован так, чтобы реализовать также интерфейс `Map`. То есть класс `Hashtable` интегрирован в инфраструктуру коллекций. Он подобен классу `HashMap`, но синхронизирован.

Подобно классу `HashMap`, класс `Hashtable` сохраняет пары "ключ-значение" в хеш-таблице. Однако ни ключи, ни значения не могут быть пусты. При использовании класса `Hashtable` вы указываете объект, который служит ключом, и значение, которое хотите связать с этим ключом. Ключ затем хешируется, а результирующий хеш-код используется в качестве индекса, по которому значение сохраняется в таблице.

Класс `Hashtable` был сделан обобщенным в JDK 5. Он объявлен следующим образом.

```
class Hashtable<K, V>
```

Здесь *K* указывает тип ключей, а *V* — тип значений.

Хеш-таблица может только сохранять объекты, которые переопределяют методы `hashCode()` и `equals()`, определенные в классе `Object`. Метод `hashCode()` должен вычислять и возвращать хеш-код объекта. Конечно, метод `equals()` должен сравнивать два объекта. К счастью, многие из встроенных классов Java уже реализуют метод `hashCode()`. Так, например, наиболее общий класс `Hashtable` использует в качестве ключа объект `String`. Класс `String` реализует и метод `hashCode()`, и метод `equals()`.

Конструкторы класса `Hashtable` показаны ниже.

```
Hashtable()
Hashtable(int размер)
```

```
Hashtable(int размер, float коэффЗаполнения)
Hashtable(Map<? extends K, ? extends V> m)
```

Первая версия — стандартный конструктор. Вторая версия создает хеш-таблицу, имеющую начальный размер, указанный параметром *размер*. (Размер по умолчанию — 11.) Третья версия создает хеш-таблицу с начальным размером *размер* и коэффициентом заполнения, заданным параметром *коэффЗаполнения*. Этот коэффициент лежит в пределах от 0,0 до 1,0 и определяет, насколько полной должна быть таблица, прежде чем она будет расширена. Точнее, когда число элементов превышает емкость, умноженную на коэффициент заполнения, хеш-таблица расширяется. Если вы не указываете коэффициент заполнения, используется значение по умолчанию 0,75. И наконец, четвертая версия создает хеш-таблицу, инициализированную элементами из коллекции *m*. По умолчанию используется коэффициент заполнения, равный 0,75.

В дополнение к методам, определенным в интерфейсе `Map`, который теперь реализует класс `Hashtable`, последний определяет также унаследованные методы, перечисленные в табл. 17.22. При попытке использования пустого ключа некоторые методы передают исключение `NullPointerException`.

Таблица 17.22. Унаследованные методы, определенные в классе `Hashtable`

Метод	Описание
`void clear()`	Сбрасывает и очищает хеш-таблицу
`Object clone()`	Возвращает дубликат вызывающего объекта
`boolean contains(Object значение)`	Возвращает значение `true`, если некоторое значение, эквивалентное *значение*, существует в хеш-таблице. Возвращает значение `false`, если значение не найдено
`boolean containsKey(Object ключ)`	Возвращает значение `true`, если некоторый ключ, эквивалентный *ключ*, существует в хеш-таблице. *Возвращает значение `false`, если ключ не найден*
`boolean containsValue(Object значение)`	Возвращает значение `true`, если некоторое значение, эквивалентное *значение*, существует в хеш-таблице. Возвращает значение `false`, если значение не найдено
`Enumeration<V> elements()`	Возвращает перечисление значений, содержащихся в хеш-таблице
`V get(Object ключ)`	Возвращает объект, который содержит значение, ассоциированное с *ключ*. Если *ключ* не найден в хеш-таблице, возвращается значение `null`
`boolean isEmpty()`	Возвращает значение `true`, если хеш-таблица пуста. Возвращает значение `false`, если она содержит хоть одно значение
`Enumeration<K> keys()`	Возвращает перечисление ключей, содержащихся в хеш-таблице
`V put(K ключ, V значение)`	Вставляет в хеш-таблицу ключ и значение. Возвращает значение `null`, если *ключ* не был найден в таблице на момент вставки. В противном случае возвращает предыдущее значение, ассоциированное с этим ключом
`void rehash()`	Увеличивает размер хеш-таблицы и повторно хеширует все ее ключи
`V remove(Object ключ)`	Удаляет *ключ* из хеш-таблицы. Возвращает значение, ассоциированное с *ключ*. Если *ключ* не найден в хеш-таблице, возвращается значение `null`

Окончание табл. 17.22

Метод	Описание
`int size()`	Возвращает количество элементов в хеш-таблице
`String toString()`	Возвращает строковый эквивалент хеш-таблицы

Следующий пример представляет переработанную версию программы управления банковскими счетами, показанную ранее, с применением класса `Hashtable` для хранения имен депозиторов и их текущих балансов.

```
// Демонстрация применения 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("Джон Доу", 3434.34);
        balance.put("Том Смит", 123.22);
        balance.put("Джейн Бейкер", 1378.00);
        balance.put("Тод Холл", 99.22);
        balance.put("Ральф Смит", -19.08);

        // Показать все счета в хеш-таблице.
        names = balance.keys();
        while(names.hasMoreElements()) {
            str = names.nextElement();
            System.out.println(str + ": " + balance.get(str));
        }
        System.out.println();
        // Добавить 1,000 на счет Джона Доу.
        bal = balance.get("Джон Доу");
        balance.put("Джон Доу", bal+1000);
        System.out.println("Новый баланс Джона Доу: " +
                           balance.get("Джон Доу"));
    }
}
```

Вывод этой программы показан ниже.

```
Тод Холл: 99.22
Ральф Смит: -19.08
Джон Доу: 3434.34
Джейн Бейкер: 1378.0
Том Смит: 123.22

Новый баланс Джона Доу: 4434.34
```

Одно существенное замечание: подобно карте, класс `Hashtable` не поддерживает напрямую итераторы. То есть предыдущая программа использует перечисление для отображения содержимого объекта `balance`. Однако вы можете получить представления хеш-таблицы в виде наборов (`Set`), которые допускают использование итераторов. Чтобы сделать это, просто воспользуйтесь одним из методов представления коллекций, определенных в интерфейсе `Map`, таким как метод `entrySet()` или `keySet()`. Например, вы можете получить представление в виде

набора всех ключей и перебрать его с применением либо итератора, либо усовершенствованного цикла for. Ниже представлена переработанная версия программы, которая демонстрирует эту технику.

```
// Применение 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("Джон Доу", 3434.34);
        balance.put("Том Смит", 123.22);
        balance.put("Джейн Бейкер", 1378.00);
        balance.put("Тод Холл", 99.22);
        balance.put("Ральф Смит", -19.08);

        // Отобразить все счета в хеш-таблице.
        // Для начала получить ключи в виде набора.
        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();

        // Добавить 1 000 на счет Джона Доу.
        bal = balance.get("Джон Доу");
        balance.put("Джон Доу", bal+1000);
        System.out.println("Новый баланс Джона Доу: " +
                           balance.get("Джон Доу"));
    }
}
```

Класс Properties

Класс Properties (свойства) — это подкласс класса Hashtable. Он служит для поддержки списков значений, в которых ключами являются объекты класса String, а значениями — также объекты класса String. Класс Properties используется многими другими классами Java. Так, например, это тип объекта, возвращаемого методом System.getProperties(), когда извлекаются переменные окружения. Хотя сам класс Properties не является обобщенным, некоторые его методы используют обобщенный синтаксис.

Класс Properties определяет следующую переменную экземпляра.

```
Properties defaults;
```

Эта переменная содержит список свойств по умолчанию, ассоциированных с объектом класса Properties.

В классе Properties определены следующие конструкторы.

```
Properties()
Properties(Properties свойствоПоУмолчанию)
```

Первая версия создает объект класса Properties, не имеющий значений по умолчанию. Вторая создает объект, используя *свойствоПоУмолчанию* в качестве значений по умолчанию. В обоих случаях список свойств пуст.

В дополнение к методам, унаследованным классом Properties от класса Hashtable, этот класс определяет методы, перечисленные в табл. 17.23. Класс Properties также имеет один нежелательный метод save(). Он был заменен методом store(), поскольку метод save() некорректно обрабатывал ошибки.

Таблица 17.23. Методы, определенные в классе Properties

Метод	Описание
String getProperty(String *ключ*)	Возвращает значение, ассоциированное с ключом *ключ*. Если ключа *ключ* нет ни в самом списке свойств, ни в списке свойств по умолчанию, то возвращается значение null
String getProperty(String *ключ*, String *свойствоПоУмолчанию*)	Возвращает значение, ассоциированное с *ключ*. Если ключа *ключ* нет ни в самом списке свойств, ни в списке свойств по умолчанию, то возвращается *свойствоПоУмолчанию*
void list(PrintStream *выходнойПоток*)	Посылает список свойств в выходной поток, связанный с *выходнойПоток*
void list(PrintWriter *выходнойПоток*)	Посылает список свойств в выходной поток, связанный с *выходнойПоток*
void load(InputStream *входнойПоток*) throws IOException	Загружает список свойств из входного потока, связанного с *входнойПоток*
void load(Reader *входнойПоток*) throws IOException	Загружает список свойств из входного потока, связанного с *входнойПоток*
void loadFromXML(InputStream *входнойПоток*) throws IOException, InvalidPropertiesFormatException	Загружает список свойств из документа XML, связанного с *входнойПоток*
Enumeration<?> propertyNames()	Возвращает перечисление ключей. Сюда включаются также ключи, найденные в списке свойств по умолчанию
Object setProperty(String *ключ*, String *значение*)	Ассоциирует *значение* с *ключ*. Возвращает предыдущее значение, ассоциированное с ключом *ключ*, либо значение null, если такой ассоциации не найдено
void store(OutputStream *выходнойПоток*, String *описание*) throws IOException	После записи строки, указанной в *описание*, список свойств записывается в поток, связанный с *выходнойПоток*
void store(Writer *выходнойПоток*, String *описание*) throws IOException	После записи строки, указанной в *описание*, список свойств записывается в поток, связанный с *выходнойПоток*

Окончание табл. 17.23

Метод	Описание
`void storeToXML(OutputStream` *выходнойПоток*`, String` *описание*`) throws IOException`	После записи строки, указанной в *описание*, список свойств записывается в документ XML, связанный с *выходнойПоток*
`void storeToXML(OutputStream` *выходнойПоток*`, String` *описание*`, String` *enc*`)`	Список свойств и строка, указанная в *описание*, записываются в документ XML, связанный с *выходнойПоток* с применением указанной кодировки символов
`Set<String> stringPropertyNames()`	Возвращает набор ключей

Одно удобное свойство класса `Properties` — это то, что вы можете указать значения по умолчанию, которые будут возвращены, если никакое значение не ассоциировано с определенным ключом. Например, значение по умолчанию может быть указано вместе с ключом в методе `getProperty()` — как, например, `getProperty("`*имя*`", "`*значение_по_умолчанию*`")`. Если значение "*имя*" не найдено, возвращается "*значение_по_умолчанию*". При создании объекта класса `Properties` вы можете передать ему другой экземпляр класса `Properties` в качестве списка свойств по умолчанию для нового экземпляра. В этом случае, если вы вызываете метод `getProperty("foo")` для данного объекта класса `Properties` и "`foo`" не существует, Java ищет его в объекте класса `Properties` по умолчанию. Это позволяет иметь произвольное количество уровней вложения свойств по умолчанию.

В следующем примере демонстрируется применение класса `Properties`. В нем создается список свойств, в котором ключами являются названия штатов, а значениями — названия столиц. Обратите внимание на то, что попытка найти столицу Флориды включает значение по умолчанию.

```
class PropDemo {
    public static void main(String args[]) {
        Properties capitals = new Properties();
        capitals.put("Иллинойс", "Спрингфилд");
        capitals.put("Миссури", "Джефферсон-Сити");
        capitals.put("Вашингтон", "Олимпия");
        capitals.put("Калифорния", "Сакраменто");
        capitals.put("Индиана", "Индианаполис");

        // Получить набор ключей.
        Set<?> states = capitals.keySet();

        // Показать все штаты и столицы.
        for(Object name : states)
            System.out.println("Столица штата " + name + " - " +
                               capitals.getProperty((String)name) +
                               ".");
        System.out.println();

        // Поиск штата, не содержащегося в списке — с указанием
        // умолчания.
        String str = capitals.getProperty("Флорида", "не найдена");
        System.out.println("Столица Флориды " + str + ".");
    }
}
```

Вывод этой программы показан ниже.

```
Столица штата Миссури - Джефферсон-Сити.
Столица штата Иллинойс - Спрингфилд.
```

```
Столица штата Индиана - Индианаполис.
Столица штата Калифорния - Сакраменто.
Столица штата Вашингтон - Олимпия.

Столица Флориды не найдена.
```

Поскольку Флорида не содержится в списке, используется значение по умолчанию.

Хотя это исключительно правильно — использовать значения по умолчанию при вызове метода `getProperty()`, как показано в предыдущем примере, существует лучший способ управления значениями по умолчанию для большинства приложений, имеющих дело со списками свойств. Для большей гибкости задавайте список свойств по умолчанию при создании объекта класса `Properties`. Если нужный ключ в главном списке на найден, поиск производится в списке по умолчанию. Например, ниже представлена слегка измененная версия предыдущей программы с применением списка штатов по умолчанию. Теперь, когда ищется столица Флориды, она будет найдена в списке по умолчанию.

```
// Использование списка свойств по умолчанию.
import java.util.*;
class PropDemoDef {
    public static void main(String args[]) {
        Properties defList = new Properties();
        defList.put("Флорида", "Тэлесси");
        defList.put("Висконсин", "Мэдисон");
        Properties capitals = new Properties(defList);
        capitals.put("Иллинойс", "Спрингфилд");
        capitals.put("Миссури", "Джефферсон-Сити");
        capitals.put("Вашингтон", "Олимпия");
        capitals.put("Калифорния", "Сакраменто");
        capitals.put("Индиана", "Индианаполис");

        // Получить набор ключей.
        Set<?> states = capitals.keySet();

        // Показать все штаты и столицы.
        for(Object name : states)
            System.out.println("Столица штата " + name + " - " +
                               capitals.getProperty((String)name) +
                               ".");
        System.out.println();

        // Теперь Флорида будет найдена в списке по умолчанию.
        String str = capitals.getProperty("Флорида");
        System.out.println("Столица Флориды - " + str + ".");
    }
}
```

Использование методов `store()` и `load()`

Один из наиболее удобных аспектов класса `Properties` в том, что информация, содержащаяся в объекте класса `Properties`, может быть легко сохранена и загружена с диска методами `store()` и `load()`. В любой момент вы можете записать объект класса `Properties` в поток либо прочесть его обратно. Это делает списки свойств особенно удобными для реализации простых баз данных. Например, следующая программа использует список свойств для создания простого телефонного справочника, хранящего имена и номера телефонов. Чтобы най-

ти номер лица, вы вводите имя. Программа использует методы `store()` и `load()` для сохранения и чтения списка. Когда эта программа выполняется, вначале она пытается загрузить список из файла по имени `phonebook.dat`. Если этот файл существует, он загружается. Затем вы можете добавлять новые значения в список. Если вы это делаете, список сохраняется при завершении программы. Обратите внимание на то, насколько компактный код требуется для реализации маленькой, но функциональной компьютеризованной телефонной книги.

```
/* Простая база данных телефонных номеров, использующая списки 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));
        String name, number;
        FileInputStream fin = null;
        boolean changed = false;

        // Попытка открыть файл phonebook.dat.
        try {
            fin = new FileInputStream("phonebook.dat");
        } catch(FileNotFoundException e) {
            // Игнорировать отсутствующий файл
        }

        /*Если телефонная книга уже существует, загрузить существующие
телефонные номера.*/
        try {
            if(fin != null) {
                ht.load(fin);
                fin.close();
            }
        } catch(IOException e) {
            System.out.println("Ошибка чтения файла.");
        }
        // Разрешить пользователю вносить новые имена и номера телефонов.
        do {
            System.out.println("Введите имя" +
                               " ('выход' для останова): ");
            name = br.readLine();
            if(name.equals("выход")) continue;
            System.out.println("Введите номер: ");
            number = br.readLine();
            ht.put(name, number);
            changed = true;
        } while(!name.equals("выход"));
        // Если телефонная книга изменилась, сохранить ее.
        if(changed) {
            FileOutputStream fout = new
                                    FileOutputStream("phonebook.dat");
            ht.store(fout, "Телефонная книга");
            fout.close();
        }
        // Искать номер по имени.
        do {
            System.out.println("Введите имя для поиска" +
                               " ('выход' для останова): ");
```

```
            name = br.readLine();
            if(name.equals("выход")) continue;
            number = (String) ht.get(name);
            System.out.println(number);
        } while(!name.equals("выход"));
    }
}
```

Заключительные соображения по поводу коллекций

Инфраструктура коллекций предлагает вам, как программисту, мощный набор тщательно спроектированных решений для некоторых наиболее часто встречающихся программистских задач. Теперь, когда инфраструктура коллекций стала обобщенной, она может применяться с поддержкой полной безопасности типов, что способствует ее дальнейшему развитию. Рассмотрите возможность применения коллекций в следующий раз, когда вам понадобится сохранять и извлекать информацию. Помните, что коллекции не предназначены только для "крупных задач" вроде корпоративных баз данных, списков почтовых рассылок либо систем инвентаризации. Они также эффективны для решения небольших задач. Например, коллекция класса `TreeMap` может отлично подойти для хранения структуры каталогов или наборов файлов. Класс `TreeSet` может оказаться довольно удобным для хранения информации по управлению проектом. В общем случае, типы проблем, при решении которых средствами коллекций можно получить существенный выигрыш, ограничиваются только вашим воображением.

ГЛАВА 18

Пакет java.util: прочие служебные классы

В этой главе продолжается обсуждение пакета `java.util`, рассматриваются классы и интерфейсы, которые не являются частью инфраструктуры коллекций. Сюда относятся классы, разбивающие строки на лексемы, работающие с датами, создающие случайные числа, связывающие ресурсы и наблюдающие за событиями. Также описываются новые классы `Formatter` и `Scanner`, которые облегчают чтение и запись форматированных данных. И наконец, вкратце упоминаются вложенные пакеты `java.util`.

Класс StringTokenizer

Обработка текста зачастую предполагает разбор форматированной входной строки. *Разбор* (parsing) – это разделение текста на набор дискретных составных частей, или *лексем* (token), представляющих определенные последовательности, которые могут иметь некоторое семантическое значение. Класс `StringTokenizer`, представляющий первый этап в процессе разбора, часто называют лексическим анализатором или сканером. Этот класс реализует интерфейс `Enumeration`. Таким образом, получая входную строку, вы можете перебрать содержащиеся в ней индивидуальные лексемы с помощью класса `StringTokenizer`.

Чтобы использовать класс `StringTokenizer`, вы указываете входную строку и строку, содержащую разделители. *Разделители* (delimiters) – это символы, разделяющие лексемы. Каждый символ в строке разделителей рассматривается как допустимый разделитель – например, строка `" , ; : "` устанавливает в качестве разделителей запятую, точку с запятой и двоеточие. Набор разделителей по умолчанию состоит из пробельных символов: пробела, знака табуляции, перевода строки и возврата каретки.

Конструкторы класса `StringTokenizer` показаны ниже.

```
StringTokenizer(String строка)
StringTokenizer(String строка, String разделители)
StringTokenizer(String строка, String разделители, boolean
разделителиКакЛексемы)
```

Во всех трех версиях *строка* – это строка, которая будет разделена на части. В первой версии используются разделители по умолчанию. Во второй и третьей версиях *разделители* – это строка, задающая разделители. В третьей версии, если параметр *разделителиКакЛексемы* содержит значение `true`, сами разделители возвращаются в качестве отдельных лексем при разборе строки. В противном случае разделители не возвращаются. Разделители также не возвращаются в первых двух формах.

Однажды создав объект класса StringTokenizer, можно использовать его метод nextToken() для извлечения последовательных лексем. Метод hasMoreTokens() возвращает значение true до тех пор, пока существуют лексемы для извлечения. Поскольку класс StringTokenizer реализует интерфейс Enumeration, методы hasMoreElements() и nextElement() также реализованы, и они работают точно так же, как, соответственно, методы hasMoreTokens() и nextToken(). Методы класса StringTokenizer перечислены в табл. 18.1.

Таблица 18.1. Методы, определенные в классе StringTokenizer

Метод	Описание
int countTokens()	Используя текущий набор разделителей, метод определяет количество лексем, которые осталось разобрать и вернуть в результате
boolean hasMoreElements()	Возвращает значение true, если одна или более лексем остались в строке, в противном случае возвращает значение false
boolean hasMoreTokens()	Возвращает значение true, если одна или более лексем остались в строке, в противном случае возвращает значение false
Object nextElement()	Возвращает следующую лексему как объект класса Object
String nextToken()	Возвращает следующую лексему как объект класса String
String nextToken(String *разделители*)	Возвращает следующую лексему как объект класса Object и устанавливает строку разделителей, как указано в *разделители*

Ниже приведен пример, создающий объект класса StringTokenizer для разбора пар "ключ/значение". Последовательность наборов "ключ/значение" разделена точкой с запятой.

```
// Демонстрация применения StringTokenizer.
import java.util.StringTokenizer;
class STDemo {
    static String in = "title=Java: The Complete Reference;" +
                       "author=Schildt;" +
                       "publisher=McGraw-Hill;" +
                       "copyright=2011";
    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);
        }
    }
}
```

Вывод этой программы выглядит так.

```
title Java: The Complete Reference
author Schildt
publisher McGraw-Hill
copyright 2011
```

Класс BitSet

Этот класс создает специальный тип массива, содержащий битовые значения. При необходимости массив BitSet может увеличиваться в размере. Это делает его похожим на битовый вектор. Конструкторы класса BitSet показаны ниже.

```
BitSet()
BitSet(int размер)
```

Первая версия создает объект по умолчанию. Вторая версия позволяет указать начальный размер (т.е. количество битов, которые можно сохранить). Все биты инициируются нулями.

Класс BitSet определяет методы, представленные в табл. 18.2.

Таблица 18.2. Методы, определенные в классе BitSet

Метод	Описание
void and(BitSet *наборБит*)	Выполняет операцию логического "И" (AND) между вызывающим объектом класса BitSet и указанным в параметре *наборБит*. Результат помещается в вызывающий объект
void andNot(BitSet *наборБит*)	Для каждого бита в наборе, равного 1, сбрасывается соответствующий бит в вызывающем объекте класса BitSet
int cardinality()	Возвращает количество установленных битов в вызывающем объекте
void clear()	Устанавливает все биты в нуль
void clear(int *индекс*)	Устанавливает в нуль бит в позиции *индекс*
void clear(int *начИндекс*, int *конИндекс*)	Устанавливает в нуль биты от *начИндекс* до *конИндекс*-1
Object clone()	Дублирует вызывающий объект
boolean equals(Object *наборБит*)	Возвращает значение true, если вызывающий набор битов эквивалентен переданному в параметре *наборБит*. В противном случае возвращает значение false
void flip(int *индекс*)	Обращает бит, находящийся в позиции *индекс*
void flip(int *начИндекс*, int *конИндекс*)	Обращает биты в диапазоне от *начИндекс* до *конИндекс*-1
boolean get(int *индекс*)	Возвращает текущее значение бита в позиции *индекс*
BitSet get(int *начИндекс*, int *конИндекс*)	Возвращает объект класса BitSet, состоящий из бит от *начИндекс* до *конИндекс*-1
int hashCode()	Возвращает хеш-код вызывающего объекта
boolean intersects(BitSet *наборБит*)	Возвращает значение true, если хотя бы одна пара соответствующих битов в вызывающем объекте и *наборБит* равна 1
boolean isEmpty()	Возвращает значение true, если все биты вызывающего объекта установлены в 0
int length()	Возвращает количество битов, необходимых для того, чтобы уместить содержимое вызывающего объекта класса BitSet. Это значение определяется по положению последнего бита, равного 1

Окончание табл. 18.2

Метод	Описание
`int nextClearBit(int` *`начИндекс`*`)`	Возвращает позицию следующего сброшенного бита (т.е. следующего бита, равного 0), начиная с индекса, указанного в *`начИндекс`*
`int nextSetBit(int` *`начИндекс`*`)`	Возвращает позицию следующего установленного бита (т.е. следующего бита, равного 1) начиная с индекса, указанного в *`начИндекс`*. Если ни один бит не установлен, возвращается значение `-1`
`void or(BitSet` *`наборБит`*`)`	Выполняет операцию логического "ИЛИ" (OR) между вызывающим объектом класса `BitSet` и указанным в параметре *`наборБит`*. Результат помещается в вызывающий объект
`int previousClearBit(int` *`начИндекс`*`)`	Возвращает индекс следующего сброшенного бита (т.е. бита, равного 0) до индекса, указанного в *`начИндекс`* или равного ему. Если сброшенного бита не найдено, возвращает значение `-1`. (Добавлено в JDK 7)
`int previousSetBit(int` *`начИндекс`*`)`	Возвращает индекс следующего установленного бита (т.е. бита, равного 1) до индекса, указанного *`начИндекс`* или равного ему. Если установленного бита не найдено, возвращает значение `-1`. (Добавлено в JDK 7)
`void set(int` *`индекс`*`)`	Устанавливает бит в позиции *`индекс`* равным `1`
`void set(int` *`индекс`*`, boolean` *`v`*`)`	Устанавливает бит в позиции *`индекс`* равным значению, переданному в *`v`*. Значение `true` устанавливает бит, а значение `false` — сбрасывает
`void set(int` *`начИндекс`*`, int` *`конИндекс`*`)`	Устанавливает биты в позициях от *`начИндекс`* до *`конИндекс`*`-1` равными 1
`void set(int` *`начИндекс`*`, int` *`конИндекс`*`, boolean` *`v`*`)`	Устанавливает биты в позициях от *`начИндекс`* до *`конИндекс`*`-1` равными значению, переданному в *`v`*. Значение `true` устанавливает бит, значение `false` — сбрасывает
`int size()`	Возвращает количество битов в вызывающем объекте класса `BitSet`
`byte[] toByteArray()`	Возвращает массив типа `byte`, который содержит вызывающий объект класса `BitSet`. (Добавлено в JDK 7)
`long[] toLongArray()`	Возвращает массив типа `long`, который содержит вызывающий объект класса `BitSet`. (Добавлено в JDK 7)
`String toString()`	Возвращает строковый эквивалент вызывающего объекта класса `BitSet`
`static BitSet valueOf(byte[]` *`v`*`)`	Возвращает объект класса `BitSet`, содержащий биты из массива *`v`*. (Добавлено в JDK 7)
`static BitSet valueOf(ByteBuffer` *`v`*`)`	Возвращает объект класса `BitSet`, содержащий биты из буфера *`v`*. (Добавлено в JDK 7)
`static BitSet valueOf(long[]` *`v`*`)`	Возвращает объект класса `BitSet`, содержащий биты из массива *`v`*. (Добавлено в JDK 7)
`static BitSet valueOf(LongBuffer` *`v`*`)`	Возвращает объект класса `BitSet`, содержащий биты из буфера *`v`*. (Добавлено в JDK 7)
`void xor(BitSet` *`наборБит`*`)`	Выполняет операцию исключающего логического "ИЛИ" (XOR) над содержимым вызывающего объекта класса `BitSet` и переданного в *`наборБит`*. Результат помещается в вызывающий объект

Ниже приведен пример, демонстрирующий использование класса `BitSet`.

```
// Демонстрация применения BitSet.
import java.util.BitSet;

class BitSetDemo {
    public static void main(String args[]) {
        BitSet bits1 = new BitSet(16);
        BitSet bits2 = new BitSet(16);

        // Установить некоторые биты
        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 шаблон в bits1: ");
        System.out.println(bits1);
        System.out.println("\nНачальный шаблон в bits2: ");
        System.out.println(bits2);

        // Логическое И над битами
        bits2.and(bits1);
        System.out.println("\nbits2 AND bits1: ");
        System.out.println(bits2);

        // Логическое ИЛИ над битами
        bits2.or(bits1);
        System.out.println("\nbits2 OR bits1: ");
        System.out.println(bits2);

        // Логическое исключающее ИЛИ над битами
        bits2.xor(bits1);
        System.out.println("\nbits2 XOR bits1: ");
        System.out.println(bits2);
    }
}
```

Вывод этой программы показан ниже. Когда метод `toString()` преобразует объект класса `BitSet` в его строковый эквивалент, каждый установленный бит представляется его позицией. Сброшенные биты не показаны.

```
Начальный шаблон в bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Начальный шаблон 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:
{}
```

Класс `Date`

Класс `Date` инкапсулирует текущую дату и время. Прежде чем мы начнем изучать класс `Date`, важно отметить, что этот класс ощутимо изменился по сравне-

нию с его оригинальной версией, определенной в Java 1.0. Когда вышла версия Java 1.1, многие функции исходного класса `Date` были перемещены в классы `Calendar` и `DateFormat` и, как результат, многие исходные методы класса `Date` были объявлены нежелательными (deprecated). Поскольку эти методы из Java 1.0 не должны использоваться в новом коде, здесь они не описываются. Класс `Date` поддерживает следующие конструкторы.

```
Date()
Date(long миллисекунд)
```

Первый конструктор инициализирует объект текущей датой и временем. Второй конструктор принимает один аргумент, представляющий количество миллисекунд, прошедших с полуночи 1 января 1970 г. Методы класса `Date`, не относящиеся к нерекомендованным, перечислены в табл. 18.3. Класс `Date` также реализует интерфейс `Comparable`.

Таблица 18.3. Методы, определенные в классе `Date`

Метод	Описание
`boolean after(Date дата)`	Возвращает значение `true`, если вызывающий объект класса `Date` содержит более позднюю дату, чем указанна в *дата*. В противном случае возвращает значение `false`
`boolean before(Date дата)`	Возвращает значение `true`, если вызывающий объект класса `Date` содержит более раннюю дату, чем указанна в *дата*. В противном случае возвращает значение `false`
`Object clone()`	Дублирует вызывающий объект класса `Date`
`int compareTo(Date дата)`	Сравнивает значения вызывающего объекта и *дата*. Возвращает 0, если эти значения эквивалентны. Возвращает отрицательное значение, если вызывающий объект содержит более раннюю дату, чем *дата*. Возвращает положительное значение, если вызывающий объект содержит более позднюю дату
`boolean equals(Object дата)`	Возвращает значение `true`, если вызывающий объект класса `Date` содержит то же самое время и дату, что и указанные в *дата*. В противном случае возвращает значение `false`
`long getTime()`	Возвращает количество миллисекунд, прошедших с полуночи 1 января 1970 г.
`int hashCode()`	Возвращает хеш-код вызывающего объекта
`void setTime(long время)`	Устанавливает время и дату в значение, переданное в параметре *время*, который представляет число миллисекунд, прошедших с полночи 1 января 1970 г.
`String toString()`	Преобразует вызывающий объект класса `Date` в строку и возвращает результат

Как вы можете видеть в табл. 18.3, новые средства класса `Date` не позволяют получать индивидуальные компоненты даты и времени. Как демонстрируется в следующей программе, вы можете только получить дату и время в терминах миллисекунд либо в строковом представлении по умолчанию, возвращаемом методом `toString()`.

Чтобы получить более детальную информацию о времени и дате, используйте класс `Calendar`.

```
// Показывает время и дату, используя только методы класса Date.
import java.util.Date;
```

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

        // Создать объект Date
        Date date = new Date();

        // Отобразить дату и время с помощью toString()
        System.out.println(date);

        // Отобразить количество миллисекунд, прошедших
        // с 1 января 1970 г. по GMT
        long msec = date.getTime();
        System.out.println("Миллисекунд, прошедших с 1
                           января 1970 г. по GMT = " + msec);
    }
}
```

Пример вывода этой программы.

```
Sat Jan 01 10:27:33 CST 2011
Миллисекунд, прошедших с 1 января 1970 г. по GMT = 1293899253417
```

Класс `Calendar`

Абстрактный класс `Calendar` представляет набор методов, позволяющих преобразовывать время в миллисекундах во множество удобных компонентов. Некоторые примеры типов информации, которые могут быть представлены, — это год, месяц, день, часы, минуты и секунды. Его задача — обеспечить своим подклассам специфические функциональные возможности по интерпретации информации о времени в соответствии с собственными правилами. Это еще один аспект библиотеки классов Java, который позволяет писать программы, работающие в различных интернациональных окружениях. Примером такого подкласса может служить класс `GregorianCalendar`.

Класс `Calendar` не имеет открытых конструкторов.

Класс `Calendar` определяет несколько защищенных переменных экземпляра. Переменная `areFieldsSet` типа `boolean` указывает, были ли установлены компоненты времени. Переменная `fields` — это массив целочисленных значений, содержащих компоненты времени. Переменная `isSet` — это массив типа `boolean`, указывающий, был ли установлен специфический компонент времени. Переменная `time` типа `long` содержит текущее время объекта. Переменная `isTimeSet` типа `boolean` указывает, что было установлено текущее время.

Некоторые часто используемые методы класса `Calendar` показаны в табл. 18.4.

Таблица 18.4. Часто используемые методы класса `Calendar`

Метод	Описание
`abstract void add(int` *которая*`, int` *значение*`)`	Добавляет *значение* к компоненту времени или даты, указанному в *которая*. Чтобы отнять, добавляйте отрицательное значение. Параметр *которая* может быть одним из полей, определенных в классе `Calendar`, таких как `Calendar.HOUR`
`boolean after(Object` *объектКалендаря*`)`	Возвращает значение `true`, если вызывающий объект класса `Calendar` содержит более позднюю дату, чем *объектКалендаря*. В противном случае возвращает значение `false`

Продолжение табл. 18.4

Метод	Описание
`boolean before(Object` *`объектКалендаря`*`)`	Возвращает значение `true`, если вызывающий объект класса `Calendar` содержит более раннюю дату, чем *`объектКалендаря`*. В противном случае возвращает значение `false`
`final void clear()`	Обнуляет все компоненты времени в вызывающем объекте
`final void clear(int` *`которая`*`)`	Обнуляет компонент времени вызывающего объекта, указанный в *`которая`*
`Object clone()`	Возвращает дубликат вызывающего объекта
`boolean equals(Object` *`объектКалендаря`*`)`	Возвращает значение `true`, если вызывающий объект класса `Calendar` содержит дату, эквивалентную *`объектКалендаря`*. В противном случае возвращает значение `false`
`int get(int` *`полеКалендаря`*`)`	Возвращает значение одного компонента вызывающего объекта. Компонент указывается в параметре *`полеКалендаря`*. Примеры компонентов, которые можно запросить, — `Calendar.YEAR`, `Calendar.MONTH`, `Calendar.MINUTE` и т.п.
`static Locale[] getAvailableLocales()`	Возвращает массив объектов класса `Locale`, содержащий региональные данные, для которых в системе доступны календари
`static Calendar getInstance()`	Возвращает объект класса `Calendar` для региональных данных и часового пояса по умолчанию
`static Calendar getInstance(TimeZone` *`ЧП`*`)`	Возвращает объект класса `Calendar` для региональных данных по умолчанию и часового пояса *`ЧП`*
`static Calendar getInstance(Locale` *`регион`*`)`	Возвращает объект класса `Calendar` для региональных данных *`регион`* и часового пояса по умолчанию
`static Calendar getInstance(TimeZone` *`ЧП`*`, Locale` *`регион`*`)`	Возвращает объект класса `Calendar` для региональных данных *`регион`* и часового пояса *`ЧП`*
`final Date getTime()`	Возвращает объект класса `Date`, содержащий время, эквивалентное вызывающему объекту
`TimeZone getTimeZone()`	Возвращает часовой пояс вызывающего объекта
`final boolean isSet(int` *`которая`*`)`	Возвращает значение `true`, если указанный компонент времени установлен. В противном случае возвращает значение `false`
`void set(int` *`которая`*`, int` *`значение`*`)`	Устанавливает компонент даты или времени вызывающего объекта, указанного в параметре *`которая`*, равным значению *`значение`*. Параметр *`которая`* должен иметь значение одного из полей класса `Calendar`, такое как `Calendar.HOUR`
`final void set(int` *`год`*`, int` *`месяц`*`, int` *`деньМесяца`*`)`	Устанавливает в вызывающем объекте различные компоненты даты и времени
`final void set(int` *`год`*`, int` *`месяц`*`, int` *`деньМесяца`*`, int` *`час`*`, int` *`минута`*`)`	Устанавливает в вызывающем объекте различные компоненты даты и времени

Метод	Описание
`final void set(int год, int месяц, int деньМесяца, int час, int минута, int секунда)`	Устанавливает в вызывающем объекте различные компоненты даты и времени
`final void setTime(Date d)`	Устанавливает в вызывающем объекте различные компоненты даты и времени. Информация передается в объекте *d* класса `Date`
`void setTimeZone(TimeZone ЧП)`	Устанавливает часовой пояс для вызывающего объекта равным *ЧП*

В классе `Calendar` определены следующие целочисленные константы, которые используются, когда вы получаете или устанавливаете компоненты календаря.

ALL_STYLES	FRIDAY	PM
AM	HOUR	SATURDAY
AM_PM	HOUR_OF_DAY	SECOND
APRIL	JANUARY	SEPTEMBER
AUGUST	JULY	SHORT
DATE	JUNE	SUNDAY
DAY_OF_MONTH	LONG	THURSDAY
DAY_OF_WEEK	MARCH	TUESDAY
DAY_OF_WEEK_IN_MONTH	MAY	UNDECIMBER
DAY_OF_YEAR	MILLISECOND	WEDNESDAY
DECEMBER	MINUTE	WEEK_OF_MONTH
DST_OFFSET	MONDAY	WEEK_OF_YEAR
ERA	MONTH	YEAR
FEBRUARY	NOVEMBER	ZONE_OFFSET
FIELD_COUNT	OCTOBER	

В следующей программе демонстрируется использование нескольких методов класса `Calendar`.

```
// Демонстрация применения 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 с региональными данными
        // и часовым поясом по умолчанию.
        Calendar calendar = Calendar.getInstance();

        // Отобразить текущее время и дату.
        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 даты и времени и отобразить ее.
        calendar.set(Calendar.HOUR, 10);
        calendar.set(Calendar.MINUTE, 29);
        calendar.set(Calendar.SECOND, 22);

        System.out.print("Измененное время: ");
        System.out.print(calendar.get(Calendar.HOUR) + ":");
        System.out.print(calendar.get(Calendar.MINUTE) + ":");
        System.out.println(calendar.get(Calendar.SECOND));
    }
}
```

Пример вывода этой программы.

```
Дата: Jan 1 2011
Время: 11:24:25
Измененное время: 10:29:22
```

Класс GregorianCalendar

Класс `GregorianCalendar` — конкретная реализация класса `Calendar`, которая представляет обычный Григорианский календарь, с которым вы хорошо знакомы. Метод `getInstance()` класса `Calendar` обычно возвращает объект класса `GregorianCalendar`, инициированный текущей датой и временем согласно региональным данным и часовому поясу по умолчанию.

Класс `GregorianCalendar` определяет два поля: `AD` и `BC`. Они представляют две эры, определенные в Григорианском календаре.

Кроме того, имеется также несколько конструкторов для объектов класса `GregorianCalendar`. Стандартный конструктор, `GregorianCalendar()`, инициирует объект текущим временем и датой в региональных данных и часовом поясе по умолчанию. Есть и другие конструкторы, представленные ниже по мере возрастания специализации.

```
GregorianCalendar(int год, int месяц, int деньМесяца)
GregorianCalendar(int год, int месяц, int деньМесяца, int час,
                  int минута)
GregorianCalendar(int год, int месяц, int деньМесяца, int час,
                  int минута, int секунда)
```

Все три версии устанавливают день, месяц и год. Здесь *год* указывает год. Месяц задает *месяц*, причем нуль означает январь. День месяца — это *деньМесяца*. Первая версия устанавливает время в полночь. Вторая версия устанавливает также часы и минуты. Третья версия добавляет секунды.

Вы также можете создать объект класса `GregorianCalendar`, указав региональные данные и/или часовой пояс. Следующие конструкторы создают объекты, инициированные текущим временем и датой, используя заданный часовой пояс и/или региональные данные.

```
GregorianCalendar(Locale регион)
GregorianCalendar(TimeZone часовойПояс)
GregorianCalendar(TimeZone часовойПояс, Locale регион)
```

Класс GregorianCalendar представляет реализацию абстрактных методов класса Calendar. Кроме того, в нем определены некоторые дополнительные методы. Наиболее интересный из них — вероятно, метод isLeapYear(), который проверяет високосный год. Его форма такова.

```
boolean isLeapYear(int год)
```

Метод возвращает значение true, если *год* — високосный год, и значение false — в противном случае.

В следующей программе демонстрируется использование класса GregorianCalendar.

```
// Демонстрация применения GregorianCalendar
import java.util.*;

class GregorianCalendarDemo {
    public static void main(String args[]) {
        String months[] = {
            "Янв", "Фев", "Мар", "Апр",
            "Май", "Июн", "Июл", "Авг",
            "Сен", "Окт", "Ноя", "Дек"};
        int year;

        // Создать Григорианский календарь, инициализированный
        // текущей датой и временем с региональными данными
        // и часовым поясом по умолчанию
        GregorianCalendar gcalendar = new GregorianCalendar();

        // Отобразить текущее время и дату.
        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 - високосный ли текущий год
        if(gcalendar.isLeapYear(year)) {
            System.out.println("Текущий год високосный");
        }
        else {
            System.out.println("Текущий год не високосный");
        }
    }
}
```

Пример вывода этой программы.

```
Дата: Jan 1 2011
Время: 11:24:27
Текущий год не високосный
```

Класс TimeZone

Еще один класс, имеющий отношение ко времени, — это класс TimeZone. Он позволяет работать с часовыми поясами, смещенными относительно Гринвича (GMT), также известного как *универсальное глобальное время* (Universal Coordinated

Time — UCT). Этот класс также учитывает летнее время. Класс TimeZone поддерживает только стандартный конструктор. Примеры методов, определенных в классе TimeZone, перечислены в табл. 18.5.

Таблица 18.5. Некоторые методы класса TimeZone

Метод	Описание
`Object clone()`	Возвращает специфичную для класса TimeZone версию метода clone()
`static String[] getAvailableIDs()`	Возвращает массив строк, представляющих имена всех часовых поясов
`static String[] getAvailableIDs(int смещениеОтГринвича)`	Возвращает массив строк, представляющих имена всех часовых поясов, отстоящих на смещение *смещениеОтГринвича* относительно GMT
`static TimeZone getDefault()`	Возвращает объект класса TimeZone, который представляет часовой пояс по умолчанию, принятый на данном компьютере
`String getID()`	Возвращает имя вызывающего объекта класса TimeZone
`abstract int getOffset(int эра, int год, int месяц, int деньМесяца, int деньНедели, int миллисекунд)`	Возвращает смещение, которое должно быть добавлено к GMT, чтобы вычислить локальное время. Это значение корректируется с учетом летнего времени. Параметры метода представляют компоненты даты и времени
`abstract int getRawOffset()`	Возвращает смещение (в миллисекундах), которое должно быть добавлено к GMT, чтобы вычислить локальное время. Значение не корректируется с учетом летнего времени
`static TimeZone getTimeZone(String назвЧП)`	Возвращает объект класса TimeZone по часовому поясу, имеющему название *назвЧП*
`abstract boolean inDaylightTime(Date d)`	Возвращает значение true, если дата, представленная в *d*, относится к летнему времени в вызывающем объекте. В противном случае возвращает значение false
`static void setDefault(TimeZone ЧП)`	Устанавливает часовой пояс по умолчанию для данного хоста. *ЧП* — ссылка на объект класса TimeZone, который нужно использовать
`void setID(String назвЧП)`	Устанавливает имя часового пояса (т.е. его идентификатор) в соответствии с *назвЧП*
`abstract void setRawOffset(int миллисекунд)`	Устанавливает смещение относительно GMT в миллисекундах
`abstract boolean useDaylightTime()`	Возвращает значение true, если вызывающий объект использует летнее время. В противном случае возвращает значение false

Класс SimpleTimeZone

Класс SimpleTimeZone — это удобный подкласс класса TimeZone. Он реализует абстрактные методы класса TimeZone и позволяет работать с часовыми поясами в Григорианском календаре. Этот класс также учитывает летнее время.

Класс SimpleTimeZone определяет четыре конструктора. Первый представлен ниже.

```
SimpleTimeZone(int смещениеОтГринвича, String назвЧП)
```

Этот конструктор создает объект класса SimpleTimeZone. Здесь *смещениеОтГринвича* — это смещение относительно Гринвича, а *назвЧП* — название часового пояса. Рассмотрим второй конструктор.

```
SimpleTimeZone(int смещениеОтГринвича, String идентификаторЧП,
               int лвМесяц0, int лвДнейВМесяце0,
               int лвДень0, int время0, int лвМесяц1,
               int лвДнейВМесяце1, int лвДень1, int время1)
```

Здесь смещение относительно GMT задается в *смещениеОтГринвича*. Имя часового пояса передается в *идентификаторЧП*. Начало действия летнего времени определяется параметрами *лвМесяц0*, *лвДнейВМесяце0*, *лвДень0* и *время0*. Окончание действия летнего времени задается параметрами *лвМесяц1*, *лвДнейВМесяце1*, *лвДень1* и *время1*.

Рассмотрим третий конструктор класса SimpleTimeZone.

```
SimpleTimeZone(int смещениеОтГринвича, String идентификаторЧП,
               int лвМесяц0, int лвДнейВМесяце0,
               int лвДень0, int время0, int лвМесяц1,
               int лвДнейВМесяце1, int лвДень1,
               int время1, int лвСмещение)
```

Здесь *лвСмещение* — количество миллисекунд, сохраненных переходом на летнее время.

И наконец, четвертый конструктор класса SimpleTimeZone.

```
SimpleTimeZone(int смещениеОтГринвича, String идентификаторЧП,
               int лвМесяц0, int лвДнейВМесяце0,
               int лвДень0, int время0, int время0режим,
               int лвМесяц1, int лвДнейВМесяце1, int лвДень1,
               int время1, int время1режим, int лвСмещение)
```

Здесь *время0режим* указывает режим начального времени, а *время1режим* — режим конечного времени. Ниже перечислены допустимые значения этих режимов.

STANDARD_TIME	WALL_TIME	UTC_TIME

Режим времени определяет, как интерпретируются значения времени. Значение режима по умолчанию, используемое другими конструкторами, — WALL_TIME.

Класс Locale

Класс Locale предназначен для создания объектов, каждый из которых описывает географический или культурный регион. Это один из нескольких классов, обеспечивающих возможность создания многонациональных программ. Например, форматы, применяемые для отображения дат, времен и чисел, в разных регионах отличаются.

Интернационализация — это тема, выходящая за пределы контекста настоящей книги. Однако большинство программ нуждается только в том, чтобы иметь дело с ее основами, что включает установки только текущих региональных данных.

Класс Locale определяет следующие константы, которые удобны для обращения с наиболее часто используемыми региональными данными.

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 класса `Locale` для Канады.

Вот как выглядят конструкторы класса `Locale`.

```
Locale(String язык)
Locale(String язык, String страна)
Locale(String язык, String страна, String вариант)
```

Эти конструкторы создают объект класса `Locale` для представления специфического языка, а в случае последних двух конструкторов — также и страны. Эти значения должны содержать стандартные коды стран и языков. В параметре *вариант* может быть предоставлена различная вспомогательная информация.

Класс `Locale` определяет несколько методов. Один из наиболее важных — метод `setDefault()` — показан ниже.

```
static void setDefault(Locale объектРегиона)
```

Это устанавливает используемые по умолчанию региональные данные, применяемые JVM, в параметре *объектРегиона*.

Вот еще несколько других интересных методов.

```
final String getDisplayCountry()
final String getDisplayLanguage()
final String getDisplayName()
```

Они возвращают читабельные для человека строки, которые могут быть использованы для отображения наименования страны, наименования языка и полного описания региональных данных.

Региональные данные по умолчанию можно получить методом `getDefault()`, показанным ниже.

```
static Locale getDefault()
```

Комплект JDK 7 внес существенные изменения в класс `Locale`, который поддерживает стандарт Internet Engineering Task Force (IETF) BCP 47, определяющий дескрипторы для идентификации языков, и стандарт Unicode Technical Standard (UTS) 35, определяющий *язык разметки региональных данных* (LDML). Поддержка стандартов BCP 47 и UTS 35 требует добавления в класс `Locale` некоторых средств, включая несколько новых методов и класс `Locale.Builder`. Кроме всех прочих, есть новый метод `getScript()`, который получает сценарий региона, и метод `toLanguageTag()`, который получает строку, содержащую языковой дескриптор региона. Класс `Locale.Builder` создает экземпляры класса `Locale`. Это гарантирует, что спецификация региона будет корректно оформлена, как определено стандартом BCP 47. (Конструкторы класса `Locale` не обеспечивают такую проверку.)

Классы `Calendar` и `GregorianCalendar` — это примеры классов, чувствительных к региональным данным. Классы `DateFormat` и `SimpleDateFormat` также зависят от региональных данных.

Класс `Random`

Класс `Random` представляет собой генератор псевдослучайных чисел. Они называются псевдослучайными, поскольку представляют собой просто сложные

распределенные последовательности. Класс Random определяет следующие конструкторы.

```
Random()
Random(long начальноеЗначение)
```

Первая версия создает генератор чисел, использующий уникальное начальное число. Вторая форма позволяет вам указать это число вручную.

Если вы инициируете объект класса Random начальным числом, то этим определяете начальную точку случайной последовательности. Если вы используете одно и то же начальное число для инициализации разных объектов класса Random, то получите от каждого из них одинаковые случайные последовательности. Если вы хотите получить разные последовательности, инициируйте объекты разными числами. Один из способов сделать это — использовать текущее время в качестве инициирующего значения для объекта класса Random. Такой подход уменьшит вероятность получения повторяющихся последовательностей.

Открытые методы класса Random перечислены в табл. 18.6.

Таблица 18.6. Методы, определенные в классе Random

Метод	Описание
`boolean nextBoolean()`	Возвращает следующее случайное значение типа boolean
`void nextBytes(byte значения[])`	Заполняет массив *значения* случайно созданными значениями
`double nextDouble()`	Возвращает следующее случайное значение типа double
`float nextFloat()`	Возвращает следующее случайное значение типа float
`double nextGaussian()`	Возвращает следующее значение гауссова случайного числа
`int nextInt()`	Возвращает следующее случайное значение типа int
`int nextInt(int n)`	Возвращает следующее случайное значение типа int в диапазоне от 0 до *n*
`long nextLong()`	Возвращает следующее случайное значение типа long
`void setSeed(long начальноеЗначение)`	Устанавливает начальное значение (т.е. начальную точку для генератора случайных чисел) равным *начальноеЗначение*

Как видите, есть семь типов случайных чисел, которые вы можете извлечь из объекта класса Random. Булевы случайные значения доступны при помощи метода nextBoolean(). Случайные байты можно получить с помощью метода nextBytes(). Целые случайные числа можно получать методом nextInt(). Случайные целочисленные длинные, равномерно распределенные по диапазону допустимых значений, выдает метод nextLong(). Методы nextFloat() и nextDouble() возвращают случайные значения типа float и double, равномерно распределенные между 0,0 и 1,0. И наконец, метод nextGaussian() возвращает значение типа double, центрированное по 0,0 со стандартным отклонением в 1,0. Это то, что известно под названием *кривая нормального распределения.*

Ниже приведен пример, демонстрирующий последовательность, создаваемую методом nextGaussian(). Он получает 100 случайных гауссовых значений и усредняет их. Программа также подсчитывает количество значений, попадающих в два стандартных отклонения — плюс или минус, используя инкремент 0,5 для каждой категории. Результат отображается графически на экране.

```
// Демонстрирует случайные гауссовы значения.
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 всех значений: " + (sum/100));

        // отобразить кривую распределения
        for(int i=0; i<10; i++) {
            for(int x=bell[i]; x>0; x--)
                System.out.print("*");
            System.out.println();
        }
    }
}
```

Ниже можно видеть пример запуска этой программы. Как видите, получается колоколоподобное распределение чисел.

```
Среднее всех значений: 0.0702235271133344
**
*******
******
***************
*****************
****************
************
**********
********
***
```

Класс `Observable`

Класс `Observable` служит для создания подклассов, за которыми могут наблюдать остальные части вашей программы. Когда с объектом такого подкласса происходят изменения, наблюдающие классы извещаются об этом. Наблюдающие классы должны реализовать интерфейс `Observer`, в котором определен метод `update()`. Этот метод вызывается, когда наблюдатель получает извещение об изменении наблюдаемого объекта.

Класс `Observable` определяет методы, показанные в табл. 18.7. Объект, который подлежит наблюдению, должен следовать двум простым правилам. Во-первых, если он изменяется, то должен вызывать метод `setChanged()`. Во-вторых, когда он готов известить наблюдателей об изменении, то должен вызвать метод `notifyObservers()`. Это заставляет наблюдающий объект (или объекты) вызывать метод `update()`. Будьте осторожны: если объект обращается к методу `notifyObservers()`, не вызвав предварительно метод `setChanged()`, то никакого действия не последует.

Наблюдаемый объект должен вызывать и метод `setChanged()`, и метод `notifyObservers()`, прежде чем будет вызван метод `update()`.

Таблица 18.7. Методы, определенные в классе `Observable`

Метод	Описание
`void addObserver(Observer` *`объект`*`)`	Добавляет *`объект`* к списку объектов, наблюдающих за вызывающим объектом
`protected void clearChanged()`	Вызов этого метода помечает вызывающий объект как "не изменявшийся"
`int countObservers()`	Возвращает количество объектов, наблюдающих за текущим объектом
`void deleteObserver(Observer` *`объект`*`)`	Удаляет *`объект`* из списка объектов, наблюдающих за вызывающим объектом
`void deleteObservers()`	Удаляет все наблюдатели вызывающего объекта
`boolean hasChanged()`	Возвращает значение `true`, если вызывающий объект модифицировался, и значение `false` – в противном случае
`void notifyObservers()`	Извещает наблюдателей о том, что вызывающий объект был изменен методом `update()`. Вторым параметром метода `update()` передается значение `null`
`void notifyObservers(Object` *`объект`*`)`	Извещает наблюдателей о том, что вызывающий объект изменялся вызовом метода `update()`. Вторым параметром метода `update()` передается *`объект`*
`protected void setChanged()`	Вызывается при изменении вызывающего объекта

Отметим, что метод `notifyObservers()` имеет две формы: с аргументом и без. Если вызвать метод `notifyObservers()` с аргументом, этот объект передается методу `update()` наблюдателя в качестве второго параметра. В противном случае методу `update()` передается значение `null`. Вы можете использовать второй параметр для передачи объекта любого типа, подходящего вашему приложению.

Интерфейс `Observer`

Чтобы организовать наблюдение за объектом, следует реализовать интерфейс `Observer`. Этот интерфейс определяет только один метод.

`void update(Observable` *`наблюдаемыйОбъект`*`, Object` *`арг`*`)`

Здесь *`наблюдаемыйОбъект`* – это объект, подлежащий наблюдению, а *`арг`* – значение, переданное методу `notifyObservers()`. Метод `update()` вызывается при изменении наблюдаемого объекта.

Пример использования интерфейса `Observer`

Ниже приведен пример, демонстрирующий работу с наблюдаемым объектом. Здесь создается класс наблюдателя по имени `Watcher`, который реализует интерфейс `Observer`. Класс, подлежащий мониторингу, называется `BeingWatched`. Он расширяет класс `Observable`. Внутри класса `BeingWatched` имеется метод `counter()`, который просто выполняет обратный отсчет от указанного значения. Он использует метод `sleep()` для ожидания 10 секунд между отсчетами. Каждый раз,

когда счетчик изменяется, вызывается метод `notifyObservers()`, которому передается в качестве аргумента текущее значение счетчика. Это заставляет вызывать метод `update()` внутри класса `Watcher`, который отображает текущее значение счетчика. Внутри метода `main()` создаются наблюдающий и наблюдаемый объекты классов `Watcher` и `BeingWatched` под именами, соответственно, `observing` и `observed`. Затем объект `observing` добавляется к списку наблюдателей для объекта `observed`. Это означает, что метод `observing.update()` будет вызываться каждый раз, когда метод `counter()` вызывает метод `notifyObservers()`.

```
/* Демонстрация применения класса Observable
   и интерфейса Observer.
*/
import java.util.*;

// Класс-наблюдатель.
class Watcher implements Observer {
    public void update(Observable obj, Object arg) {
        System.out.println("update() вызван, count равен " +
                           ((Integer)arg).intValue());
    }
}

// Это - наблюдаемый класс.
class BeingWatched extends Observable {
    void counter(int period) {
        for( ; period >=0; period--) {
            setChanged();
            notifyObservers(new Integer(period));
            try {
                Thread.sleep(100);
            } catch(InterruptedException e) {
                System.out.println("Ожидание прервано");
            }
        }
    }
}

class ObserverDemo {
    public static void main(String args[]) {
        BeingWatched observed = new BeingWatched();
        Watcher observing = new Watcher();
        /* Добавить наблюдателя в список наблюдателей наблюдаемого объекта
*/
        observed.addObserver(observing);
        observed.counter(10);
    }
}
```

Вывод этой программы показан ниже.

```
update() вызван, count равен 10
update() вызван, count равен 9
update() вызван, count равен 8
update() вызван, count равен 7
update() вызван, count равен 6
update() вызван, count равен 5
update() вызван, count равен 4
update() вызван, count равен 3
update() вызван, count равен 2
update() вызван, count равен 1
update() вызван, count равен 0
```

Наблюдателями могут быть несколько объектов. Например, в следующей программе реализуются два класса наблюдателя, и объекты каждого класса добавляются к списку наблюдателей класса `BeingWatched`. Второй наблюдатель ожидает, пока счетчик достигнет нулевого значения, после чего подает звуковой сигнал.

```
/* Объект может наблюдаться одним
   или более наблюдателями.
*/
import java.util.*;

// Первый класс-наблюдатель.
class Watcher1 implements Observer {
    public void update(Observable obj, Object arg) {
        System.out.println("update() вызван, count равен " +
                           ((Integer)arg).intValue());
    }
}

// Второй класс-наблюдатель.
class Watcher2 implements Observer {
    public void update(Observable obj, Object arg) {
        // По окончании выдать звуковой сигнал
        if(((Integer)arg).intValue() == 0)
            System.out.println("Готово" + '\7');
    }
}

// Наблюдаемый класс.
class BeingWatched extends Observable {
    void counter(int period) {
        for( ; period >=0; period--) {
            setChanged();
            notifyObservers(new Integer(period));

            try {
                Thread.sleep(100);
            } catch(InterruptedException e) {
                System.out.println("Ожидание прервано");
            }
        }
    }
}

class TwoObservers {
    public static void main(String args[]) {
        BeingWatched observed = new BeingWatched();
        Watcher1 observing1 = new Watcher1();
        Watcher2 observing2 = new Watcher2();

        // Добавить оба наблюдателя
        observed.addObserver(observing1);
        observed.addObserver(observing2);
        observed.counter(10);
    }
}
```

Класс `Observable` и интерфейс `Observer` позволяют реализовывать изощренные программные архитектуры, основанные на методологии "документ-представление". Они также удобны в многопоточных программах.

Классы Timer и TimerTask

Интересным и полезным средством, предоставляемым пакетом java.util, является возможность планировать запуск задания на определенное время в будущем. Это обеспечивают классы Timer и TimerTask. Используя эти классы, вы можете создать поток, выполняющийся в фоновом режиме и ожидающий заданное время. Когда время истечет, задача, связанная с этим потоком, будет запущена. Различные параметры позволяют запланировать задачу на повторяющийся запуск либо на запуск по определенной дате. Хотя всегда существует возможность вручную создать задачу, которая будет запущена в определенное время с помощью класса Thread, все же классы Timer и TimerTask значительно упрощают этот процесс.

Классы Timer и TimerTask работают вместе. Класс Timer используется для планирования выполнения задачи. Запланированная к выполнению задача должна быть экземпляром класса TimerTask. То есть, чтобы запланировать задачу, вы сначала создаете объект класса TimerTask, а затем планируете его запуск с помощью экземпляра класса Timer.

Класс TimerTask реализует интерфейс Runnable. Это значит, что он может быть использован для создания потока выполнения. Его конструктор показан ниже.

```
TimerTask()
```

В классе TimerTask определены методы, перечисленные в табл. 18.8. Обратите внимание на то, что метод run() является абстрактным, а это означает, что он должен быть переопределен. Метод run(), определенный в интерфейсе Runnable, содержит исполняемый код. То есть простейший способ создать задачу для таймера – это расширить класс TimerTask и переопределить метод run().

Таблица 18.8. Методы, определенные в классе TimerTask

Метод	Описание
boolean cancel()	Прерывает задание. Возвращает значение true, если выполнение задания прервано. В противном случае возвращает значение false
abstract void run()	Содержит код задания таймера
long scheduledExecutionTime()	Возвращает время, на которое последний раз планировался запуск задания

Как только задача создана, она планируется для выполнения объектом класса Timer. Вот как выглядят конструкторы класса Timer.

```
Timer()
Timer(boolean потокД)
Timer(String имяПотока)
Timer(String имяПотока, boolean потокД)
```

Первая версия создает объект класса Timer и затем запускает его как обычный поток. Вторая использует поток-демон, если параметр *потокД* равен true. Поток-демон будет выполняться только до тех пор, пока выполняется остальная часть программы. Третий и четвертый конструкторы позволяют указывать имя объекта класса Timer. Методы класса Timer перечислены в табл. 18.9.

Таблица 18.9. Методы, определенные в классе Timer

Метод	Описание
void cancel()	Прерывает поток таймера
int purge()	Удаляет прерванные задания из очереди таймера

Окончание табл. 18.9

Метод	Описание
`void schedule(TimerTask задачаТ, long ожидать)`	Задание *задачаТ* планируется к выполнению через период в миллисекундах, переданный в параметре *ожидать*
`void schedule(TimerTask задачаТ, long ожидать, long повторять)`	Задание *задачаТ* планируется к выполнению через период в миллисекундах, переданный в параметре *ожидать*. Задание затем выполняется повторно периодически – каждые *повторять* миллисекунд
`void schedule(TimerTask задачаТ, Date времяЗапуска)`	Задание *задачаТ* планируется к выполнению на время, указанное в параметре *времяЗапуска*
`void schedule(TimerTask задачаТ, Date времяЗапуска, long повторять)`	Задание *задачаТ* планируется к выполнению на время, указанное в параметре *времяЗапуска*. Задание затем выполняется повторно периодически – каждые *повторять* миллисекунд
`void scheduleAtFixedRate(TimerTask задачаТ, long ожидать, long повторять)`	Задание *задачаТ* планируется к выполнению через период в миллисекундах, переданный в параметре *ожидать*. Задание затем выполняется повторно периодически – каждые *повторять* миллисекунд. Время каждого повтора задается относительно первого запуска, а не предыдущего. То есть общее время выполнения остается фиксированным
`void scheduleAtFixedRate(TimerTask задачаТ, Date времяЗапуска, long повторять)`	Задание *задачаТ* планируется к выполнению на время, указанное в параметре *времяЗапуска*. Задание затем выполняется повторно периодически – каждые *повторять* миллисекунд. Время каждого повтора задается относительно первого запуска, а не предыдущего. То есть общее время выполнения остается фиксированным

Как только объект класса `Timer` создан, запуск планируется вызовом его метода `schedule()`. Как показано в табл. 18.9, существует несколько форм метода `schedule()`, позволяющих запланировать задание разными способами.

Если вы создаете задание, не являющееся демоном, то, возможно, захотите вызвать метод `cancel()` для его прерывания при завершении программы. Если вы не сделаете этого, то ваша программа может "зависнуть" на некоторое время.

В следующей программе демонстрируется работа с классами `Timer` и `TimerTask`. В ней определяется задание таймера, метод `run()` которого отображает сообщение "Задание таймера выполняется". Это задание планируется на запуск каждые полсекунды после начальной паузы в одну секунду.

```
// Демонстрация применения Timer и TimerTask.
import java.util.*;

class MyTimerTask extends TimerTask {
    public void run() {
        System.out.println("Задание таймера выполняется.");
    }
}

class TTest {
    public static void main(String args[]) {
        MyTimerTask myTask = new MyTimerTask();
```

```
        Timer myTimer = new Timer();
        /* Устанавливает начальную паузу в 1 секунду,
           затем повторяется каждые полсекунды.
        */
        myTimer.schedule(myTask, 1000, 500);
        try {
            Thread.sleep(5000);
        } catch (InterruptedException exc) {}
        myTimer.cancel();
    }
}
```

Класс Currency

Класс Currency инкапсулирует информацию о валюте. Он не определяет конструкторов. Методы класса Currency перечислены в табл. 18.10. В следующей программе демонстрируется использование класса Currency.

```
// Демонстрация применения 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("Количество дробных разрядов по умолчанию: "+
                           c.getDefaultFractionDigits());
    }
}
```

Рассмотрим вывод этой программы.

```
Символ: $
Количество дробных разрядов по умолчанию: 2
```

Таблица 18.10. Методы, определенные в классе Currency

Метод	Описание
`static Set<Currency> getAvailableCurrencies()`	Возвращает набор поддерживаемых валют. (Добавлено в JDK 7)
`String getCurrencyCode()`	Возвращает код валюты в стандарте ISO 4217
`int getDefaultFractionDigits()`	Возвращает количество десятичных знаков после точки, которые обычно используются с данной валютой. Например, для доллара это будет 2 знака
`String getDisplayName()`	Возвращает название запрошенной валюты для заданного по умолчанию региона. (Добавлено в JDK 7)
`String getDisplayName(Locale` *регион*`)`	Возвращает название запрошенной валюты для заданного региона. (Добавлено в JDK 7)
`static Currency getInstance(Locale` *объектРегиона*`)`	Возвращает объект класса Currency для региональных данных, указанных параметром *объектРегиона*

Окончание табл. 18.10

Метод	Описание
`static Currency getInstance(String код)`	Возвращает объект класса `Currency`, ассоциированный с кодом валюты, переданным в параметре *код*
`int getNumericCode()`	Возвращает числовой код (как определено стандартом ISO 4217) для запрошенной валюты. (Добавлено в JDK 7)
`String getSymbol()`	Возвращает символ валюты (такой, как $) для вызывающего объекта
`String getSymbol(Locale объектРегиона)`	Возвращает символ валюты (такой, как $) для региональных данных, указанных параметром *объектРегиона*
`String toString()`	Возвращает код валюты вызывающего объекта

Класс `Formatter`

В центре системы поддержки форматированного вывода находится класс `Formatter`. Он предлагает *преобразования формата*, позволяющие отображать числа, строки, время и даты практически в любом виде по вашему желанию. Он работает подобно функции C/C++ `printf()`, из чего следует, что если вы знакомы с языком C/C++, то изучение класса `Formatter` для вас будет очень простым. Это также упрощает преобразование кода C/C++ в Java. Если вы не знакомы с языком C/C++, все равно будет достаточно просто научиться форматировать данные.

На заметку! Несмотря на то что класс `Formatter` очень похож на функцию C/C++ `printf()`, существуют некоторые отличия и усовершенствования. Таким образом, даже если вы имеете опыт использования языка C/C++, все равно рекомендуется внимательно прочитать этот раздел.

Конструкторы класса `Formatter`

Прежде чем можно будет использовать класс `Formatter` для форматирования вывода, следует создать его объект. В общем случае объект класса `Formatter` работает, преобразуя бинарную форму данных, используемых программой, в форматированный текст. Он явно сохраняет форматированный текст в буфере, содержимое которого может быть доступно вашей программе в любой момент, когда понадобится. Можно позволить объекту класса `Formatter` создавать этот буфер автоматически или же указать его явно при создании объекта класса `Formatter`. Можно также заставить объект класса `Formatter` направлять свой буфер в файл.

Класс `Formatter` определяет много конструкторов, позволяющих создавать его объекты разными способами.

```
Formatter()
Formatter(Appendable буфер)
Formatter(Appendable буфер, Locale регион)
Formatter(String имяфайла) throws FileNotFoundException
Formatter(String имяфайла, String наборсимволов)
throws FileNotFoundException, UnsupportedEncodingException
Formatter(File выхФайл) throws FileNotFoundException
Formatter(OutputStream выхПоток)
```

Здесь параметр *буфер* указывает буфер для форматированной строки. Если буфер пуст, то объект класса `Formatter` автоматически резервирует объект класса `StringBuffer` для хранения отформатированной строки. Параметр *регион* указывает региональные данные. Если региональные данные не заданы, используются региональные данные по умолчанию. Параметр *имяфайла* определяет имя файла, который будет принимать форматированный вывод. Параметр *наборсимволов* указывает набор символов. Если не указано никакого набора символов, то используется набор символов по умолчанию. Параметр *выхФайл* представляет собой ссылку на открытый файл, который должен принимать вывод. Параметр *выхПоток* задает ссылку на выходной поток, куда будет направлен вывод. Когда используется файл, вывод также пишется в файл.

Возможно, наиболее широко применяется первый конструктор, который не имеет параметров. Он автоматически использует региональные данные по умолчанию и резервирует объект класса `StringBuffer` для хранения отформатированного вывода.

Методы класса `Formatter`

Класс `Formatter` определяет методы, перечисленные в табл. 18.11.

Таблица 18.11. Методы, определенные в классе `Formatter`

Метод	Описание
`void close()`	Закрывает вызываемый объект класса `Formatter`. Это приводит к тому, что все ресурсы, используемые объектом, освобождаются. После закрытия объекта класса `Formatter` его больше нельзя использовать. Попытка использования закрытого объекта класса `Formatter` приводит к передаче исключения `FormatterClosedException`
`void flush()`	Сбрасывает буфер формата. Это приводит к тому, что весь вывод, находящийся в буфере, записывается в адресат. В основном, используется для объекта класса `Formatter`, примененного к файлу
`Formatter format(String` *формСтрока*`, Object ...` *аргументы*`)`	Форматирует аргументы, переданные в *аргументы*, в соответствии со спецификаторами формата, содержащимися в *формСтрока*. Возвращает вызываемый объект
`Formatter format(Locale` *регион*`, String` *формСтрока*`, Object ...` *аргументы*`)`	Форматирует аргументы, переданные в *аргументы*, в соответствии со спецификаторами формата, содержащимися в *формСтрока*. При форматировании используются региональные данные, определенные в параметре *регион*. Возвращает вызываемый объект
`IOException ioException()`	Если лежащий в основе объект, указанный в качестве адресата, передает исключение `IOException`, возвращается это исключение. В противном случае возвращается значение `null`
`Locale locale()`	Возвращает региональные данные вызывающего объекта
`Appendable out()`	Возвращает ссылку на лежащий в основе объект, который назначен в качестве адресата для вывода
`String toString()`	Возвращает объект класса `String`, содержащий форматированный вывод

Основы форматирования

После того как вы создали объект класса `Formatter`, его можно применять для создания форматированных строк. Для этого используйте метод `format()`. Его наиболее часто используемая версия показана ниже.

`Formatter format(String` *`формСтрока`*`, Object ...` *`аргументы`*`)`

Строка *`формСтрока`* состоит из элементов двух типов. Первый тип — символы, которые просто копируются в выходной буфер. Второй тип — *спецификаторы формата*, определяющие способ, в соответствии с которым должны отображаться последующие аргументы.

В простейшей форме спецификатор формата начинается со знака процента с последующим *спецификатором преобразования*. Все спецификаторы преобразования формата состоят из единственного символа. Например, спецификатор формата для числа с плавающей точкой выглядит как `%f`. В общем случае должно быть столько аргументов, сколько есть спецификаторов формата, и соответствие аргументов устанавливается слева направо. Например, рассмотрим следующий фрагмент кода.

```
Formatter fmt = new Formatter();
fmt.format("Форматировать %s очень просто: %d %f",
           "с помощью Java", 10, 98.6);
```

Приведенный фрагмент кода создает объект класса `Formatter`, содержащий следующую строку.

```
Форматировать с помощью Java очень просто: 10 98.600000
```

В этом примере спецификаторы формата `%s`, `%d` и `%f` замещаются аргументами, следующими за строкой формата. То есть `%s` заменяется на `"с помощью Java"`, `%d` — на `10`, а `%f` — на `98.6`. Все остальные символы используются, как есть. Как вы можете ожидать, спецификатор параметра `%s` указывает строку, а спецификатор `%d` — целое число. Как уже упоминалось, спецификатор `%f` означает число с плавающей точкой.

Метод `format()` принимает широкое разнообразие спецификаторов формата, которые показаны в табл. 18.12. Обратите внимание на то, что многие спецификаторы имеют и заглавную, и строчную формы. Когда используется заглавная форма, буквы отображаются в верхнем регистре. Во всем остальном формы эквивалентны. Важно понимать, что Java проверяет каждый спецификатор формата на соответствие типу аргумента. Если аргумент не соответствует, передается исключение `IllegalFormatException`.

Таблица 18.12. Спецификаторы формата

Спецификатор формата	Применяемое преобразование
`%a` `%A`	Шестнадцатеричное с плавающей точкой
`%b` `%B`	Булево
`%c`	Символ
`%d`	Десятичное целое
`%h` `%H`	Хеш-код аргумента
`%e` `%E`	Научная нотация

Окончание табл. 18.12

Спецификатор формата	Применяемое преобразование
%f	Десятичное с плавающей точкой
%g %G	Использует либо %e, либо %f, в зависимости от того, что короче
%o	Восьмеричное целое
%n	Вставляет символ перевода строки
%s %S	Строка
%t %T	Время и дата
%x %X	Шестнадцатеричное целое
%%	Вставляет символ %

После того как отформатируете строку, можете получить ее методом `toString()`. Например, следующий оператор получает форматированную строку, содержащуюся в объекте `fmt`.

```
String str = fmt.toString();
```

Конечно, если вы просто хотите отобразить форматированную строку, нет причин вначале присваивать ее объекту класса `String`. Например, когда объект класса `Formatter` передается методу `println()`, то автоматически вызывается его метод `toString()`.

Ниже приведена короткая программа, которая собирает все вместе, демонстрируя создание и отображение отформатированной строки.

```
// Очень простой пример применения Formatter.
import java.util.*;

class FormatDemo {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        fmt.format("Форматирование %s просто %d %f", " с Java", 10,
                   98.6);

        System.out.println(fmt);
        fmt.close();
    }
}
```

Еще один момент: вы можете получить ссылку на выходной буфер, вызывая метод `out()`. Он возвращает ссылку на объект интерфейса `Appendable`.

Теперь, когда вы знакомы с общим механизмом создания форматированных строк, подробно рассмотрим каждое преобразование. Кроме того, опишем такие параметры, как выравнивание, минимальная ширина поля и точность.

Форматирование строк и символов

Для форматирования индивидуального символа используйте спецификатор `%c`. В результате соответствующий символьный аргумент будет выводиться без каких-либо модификаций. Чтобы отформатировать строку, применяйте спецификатор `%s`.

Форматирование чисел

Чтобы форматировать целые числа в десятичном формате, используйте спецификатор `%d`. Чтобы форматировать значения с плавающей точкой в десятичном формате, применяйте спецификатор `%f`. Для форматирования значений с плавающей точкой в научной нотации указывайте спецификатор `%e`. Числа, представленные в научной нотации, принимают следующую общую форму.

x.dddddde+/-yy

Спецификатор формата `%g` заставляет класс `Formatter` использовать либо спецификатор `%f`, либо `%e`, в зависимости от того, что короче. В следующей программе демонстрируется эффект спецификатора формата `%g`.

```
// Демонстрирует применение спецификатора формата %g.
import java.util.*;
class FormatDemo2 {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();
        for(double i=1000; i < 1.0e+10; i *= 100) {
            fmt.format("%g ", i);
            System.out.println(fmt);
        }
        fmt.close();
    }
}
```

Эта программа дает такой вывод.

```
1000.000000
1000.000000 100000.000000
1000.000000 100000.000000 1.000000e+07
1000.000000 100000.000000 1.000000e+07 1.000000e+09
```

Вы можете отображать целые числа в восьмеричном или шестнадцатеричном формате, используя соответственно спецификаторы `%o` или `%x`. Например, следующий фрагмент кода

```
fmt.format("Шестнадцатеричное: %x, восьмеричное: %o", 196, 196);
```

создает такой вывод.

```
Шестнадцатеричное: c4, восьмеричное: 304
```

Вы можете отображать числа с плавающей точкой в шестнадцатеричном формате, используя спецификатор `%a`. Формат, создаваемый спецификатором `%a`, на первый взгляд может показаться странным. Дело в том, что его представление использует форму, подобную научной нотации, состоящей из мантиссы и экспоненты, обе — в шестнадцатеричном формате. Вот как выглядит общий формат.

0x1.sigpexp

Здесь *sig* содержит дробную часть мантиссы, а *exp* — экспоненту. Символ `p` указывает начало экспоненты. Например, следующий вызов метода

```
fmt.format("%a", 123.123);
```

создает такой вывод.

```
0x1.ec7df3b645a1dp6
```

Форматирование времени и даты

Одно из наиболее мощных преобразований задается с помощью спецификатора формата `%t`. Он позволяет форматировать информацию о времени и дате.

Спецификатор `%t` работает несколько иначе, чем другие, поскольку требует применения суффиксов для описания частичного или точного формата времени и даты. Суффиксы перечислены в табл. 18.13. Например, чтобы отобразить минуты, следует использовать спецификатор `%tM`, где `M` означает минуты в двухсимвольном поле. Аргументы, относящиеся к спецификатору `%t`, должны иметь тип `Calendar`, `Date`, `Long` или `long`.

Таблица 18.13. Суффиксы формата времени и даты

Суффикс	Заменяется на
a	Сокращенное название дня недели
A	Полное название дня недели
b	Сокращенное название месяца
B	Полное название месяца
c	Стандартная строка даты и времени, отформатированная как *день месяц дата чч::мм:сс пояс год*
C	Первые два знака года
d	День месяца как десятичное число (01–31)
D	месяц/день/год
e	День месяца как десятичное число (1–31)
F	год-месяц-день
h	Сокращенное название месяца
H	Часы (от 00 до 23)
I	Часы (от 01 до 12)
j	День года как десятичное число (от 001 до 366)
k	Часы (от 0 до 23)
l	Часы (от 1 до 12)
L	Миллисекунды (от 000 до 999)
m	Месяц как десятичное число (от 01 до 13)
M	Минуты как десятичное число (от 00 до 59)
N	Наносекунды (от 000000000 до 999999999)
p	Локальный эквивалент AM или PM в нижнем регистре
Q	Миллисекунды, прошедшие с 01/01/1970
r	*чч:мм* (12-часовой формат)
R	*чч:мм* (24-часовой формат)
S	Секунды (от 00 до 60)
s	Секунды, прошедшие с 01/01/1970 UTC
T	*чч:мм:сс* (24-часовой формат)
y	Годы в десятичных числах без века (от 00 до 99)
Y	Годы в десятичных числах, включая век (от 0001 до 9999)
z	Смещение от UTC
Z	Наименование часового пояса

Ниже приведена программа, демонстрирующая применение некоторых форматов.

```
// Форматирования времени и даты.
import java.util.*;

class TimeDateFormat {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();

        // Отобразить стандартный 12-часовой формат.
        fmt.format("%tr", cal);
        System.out.println(fmt);

        // Отобразить полную информацию о дате и времени.
        fmt = new Formatter();
        fmt.format("%tc", cal);
        System.out.println(fmt);

        // Отобразить только часы и минуты.
        fmt = new Formatter();
        fmt.format("%tl:%tM", cal, cal);
        System.out.println(fmt);

        // Отобразить название и номер месяца.
        fmt = new Formatter();
        fmt.format("%tB %tb %tm", cal, cal, cal);
        System.out.println(fmt);
        fmt.close();
    }
}
```

Рассмотрим пример вывода.

```
09:17:15 AM
Mon Jan 01 09:17:15 CST 2007
9:17
January Jan 01
```

Спецификаторы %n и %%

Спецификаторы формата %n и %% отличаются от других тем, что они не соответствуют аргументу. Вместо этого они просто представляют собой управляющие последовательности, которые вставляют символ в выходную последовательность. Спецификатор %n вставляет перевод строки, а спецификатор %% — знак процента. Ни один из этих символов не может быть введен непосредственно в форматирующую строку. Конечно, вы можете также использовать стандартную управляющую последовательность \n, чтобы вставить знак перевода строки.

Ниже приведен пример, демонстрирующий использование спецификаторов формата %n и %%.

```
// Демонстрация применения спецификаторов формата %n и %%.
import java.util.*;
class FormatDemo3 {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        fmt.format("Копирование файла%nПеремещение на %d%% завершено",
                   88);
```

```
        System.out.println(fmt);

        fmt.close();
    }
}
```

Программа отображает следующий вывод.

```
Копирование файла
Перемещение на 88% завершено
```

Указание минимальной ширины поля

Целое число, помещенное между символом % и кодом преобразования формата, выступает в качестве *спецификатора минимальной ширины*, который дополняет вывод пробелами, чтобы обеспечивать заданную минимальную длину. Если строка или число получаются длиннее этого заданного минимума, они будут напечатаны полностью. По умолчанию дополнение осуществляется пробелами. Если вы хотите дополнять нулями, поместите 0 перед спецификатором ширины поля. Например, %05d дополнит число, состоящее из менее чем 5 разрядов, нулями — чтобы его общая ширина была равна пяти. Спецификатор ширины поля может применяться вместе со всеми спецификаторами формата, кроме %n.

В следующей программе демонстрируется использование спецификатора минимальной ширины поля за счет добавления его к спецификатору преобразования %f.

```
// Демонстрация применения спецификатора 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();
    }
}
```

Эта программа создает следующий вывод.

```
|10.123450|
|   10.123450|
|00010.123450|
```

Первая строка отображает число 10,12345 с шириной по умолчанию. Вторая выводит это значение в 12-символьном поле. Третья строка отображает значение в 12-символьном поле, дополняя его предваряющими нулями.

Минимальный модификатор ширины поля часто используется для создания таблиц, состоящих из строк и столбцов. Например, следующая программа выдает таблицу квадратов и кубов чисел от 1 до 10.

```
// Создает таблицу 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();
        }
    }
}
```

Ее вывод показан ниже.

```
   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 применим к спецификаторам формата `%f`, `%e`, `%g` и `%s`. Он следует за спецификатором минимальной ширины поля (если таковой имеется) и состоит из точки с последующим целым числом. Его конкретное значение зависит от типа данных, к которому он применяется.

Когда вы применяете спецификатор точности к данным с плавающей точкой с использованием спецификаторов преобразования `%f` или `%e`, то он определяет количество отображаемых десятичных разрядов. Например, `%10.4f` отображает число, по меньшей мере, в 10 символов шириной с четырьмя разрядами после запятой. При использовании спецификатора `%g` точность определяет количество значащих десятичных разрядов. Точность по умолчанию составляет 6 знаков после запятой.

В применении к строкам спецификатор точности задает максимальную ширину поля. Например, `%5.7s` отображает строку длиной минимум в пять символов и не превышающей семь символов. Если строка длиннее максимальной ширины, конечные символы будут усечены.

В следующей программе демонстрируется работа с модификатором точности.

```
// Демонстрация применения модификатора точности.
import java.util.*;
class PrecisionDemo {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        // Формат с 4 десятичными разрядами.
        fmt.format("%.4f", 123.1234567);
        System.out.println(fmt);
        fmt.close();

        // Формат с 2 десятичными разрядами в 16-символьном поле.
        fmt = new Formatter();
        fmt.format("%16.2e", 123.1234567);
        System.out.println(fmt);
        fmt.close();

        // Отобразить максимум 15 символов строки.
```

```
        fmt = new Formatter();
        fmt.format("%.15s",
                   "Форматировать в Java теперь очень просто.");
        System.out.println(fmt);
        fmt.close();
    }
}
```

Ниже представлен вывод этой программы.

```
123.1235
    1.23e+02
Форматировать в
```

Использование флагов формата

Класс `Formatter` распознает набор *флагов* формата, которые позволяют вам управлять различными аспектами преобразования. Все флаги формата — одиночные символы, и флаг формата следует за знаком `%` в спецификаторе формата. Флаги перечислены в табл. 18.14.

Таблица 18.14. Флаги формата

Флаг	Эффект
`-`	Выравнивание влево
`#`	Альтернативный формат преобразования
`0`	Вывод дополняется нулями вместо пробелов
пробел	Положительным числам предшествует пробел
`+`	Положительным числам предшествует знак `+`
`,`	Числовые значения, включающие групповые разделители
`(`	Отрицательные числовые значения заключены в скобки

Не все флаги применимы ко всем спецификаторам формата. В следующих разделах они объясняются более подробно.

Выравнивание вывода

По умолчанию весь вывод выравнивается вправо. Иными словами, если ширина поля больше, чем выводимые данные, то эти данные будут размещены в правой части поля. Вы можете выравнивать значения влево, поместив знак "минус" сразу после `%`. Например, `%-10.2f` выравнивает влево число с плавающей точкой с двумя разрядами после десятичной точки в пределах 10-символьного поля.

Например, рассмотрим следующую программу.

```
// Демонстрация выравнивания влево.
import java.util.*;
class LeftJustify {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        // По умолчанию выравнивается вправо
        fmt.format("|%10.2f|", 123.123);
        System.out.println(fmt);
        fmt.close();
```

```
        // А теперь влево.
        fmt = new Formatter();
        fmt.format("|%-10.2f|", 123.123);
        System.out.println(fmt);
        fmt.close();
    }
}
```

Получим такой результат.

```
|    123.12|
|123.12     |
```

Как видите, вторая строка выровнена влево в пределах 10-символьного поля.

Флаги пробела, +, 0 и (

Чтобы заставить отображать знак + перед положительными числовыми значениями, добавьте флаг +. Например,

```
fmt.format("%+d", 100);
```

порождает такую строку.

```
+100
```

Когда создаются столбцы чисел, иногда удобно выводить пробел перед положительными числами, чтобы положительные и отрицательные значения выводились в столбец. Чтобы достичь этого, добавьте флаг пробела.

```
// Демонстрация применения пробела в качестве спецификатора формата.
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();
    }
}
```

Результат показан ниже.

```
-100
 100
-200
 200
```

Обратите внимание на то, что положительные значения имеют ведущий пробел, что обеспечивает ровное расположение разрядов в столбце.

Чтобы отобразить отрицательные числовые значения в скобках вместо добавления ведущего минуса, используйте флаг (. Например,

```
fmt.format("%(d", -100);
```

дает следующий результат.

```
(100)
```

Флаг 0 заставляет дополнять вывод нулями вместо пробелов.

Флаг "запятая"

При отображении больших чисел зачастую удобно добавлять разделители групп, которыми в англоязычной среде являются запятые. Например, значение 1234567 легче читается, когда оно отформатировано в виде 1,234,567. Для добавления спецификаторов группирования служит флаг "запятая" (,). Например,

```
fmt.format("%,.2f", 4356783497.34);
```

выдает такую строку.

```
4,356,783,497.34
```

Флаг

Этот флаг может применяться к спецификаторам %o, %x, %e и %f. Для спецификаторов %e и %f флаг # обеспечивает наличие десятичной точки даже в случае, если нет дробных разрядов. Если вы перед спецификатором формата %x поставите флаг #, то шестнадцатеричное число будет выведено с префиксом 0x. Если предварить флагом # спецификатор %x, число будет выводиться с ведущим нулем.

Параметры верхнего регистра

Как упоминалось ранее, некоторые спецификаторы формата имеют версии в верхнем регистре, которые заставляют при преобразовании применять буквы верхнего регистра, где это возможно. В табл. 18.15 описывается упомянутый эффект.

Например, следующий вызов

```
fmt.format("%X", 250);
```

порождает следующую строку.

```
FA
```

Приведенный ниже вызов

```
fmt.format("%E", 123.1234);
```

дает в результате такую строку.

```
1.231234E+02
```

Таблица 18.15. Параметры верхнего регистра

Спецификатор	Эффект
%A	Заставляет шестнадцатеричные цифры от *a* до *f* отображаться в верхнем регистре, т.е. от *A* до *F*. Кроме того, префикс 0x отображается как 0X, а p – как P
%B	Переводит в верхний регистр значения true и false

Окончание табл. 18.15

Спецификатор	Эффект
%E	Заставляет символ экспоненты e отображаться в верхнем регистре
%G	Заставляет символ экспоненты e отображаться в верхнем регистре
%H	Заставляет шестнадцатеричные цифры от *a* до *f* отображаться в верхнем регистре, от *A* до *F*
%S	Переводит соответствующую спецификатору строку в верхний регистр
%T	Заставляет алфавитный вывод отображаться в верхнем регистре
%X	Заставляет шестнадцатеричные цифры от *a* до *f* отображаться в верхнем регистре, от *A* до *F*. Кроме того, префикс 0x отображается как 0X, если таковой присутствует

Использование индекса аргументов

Класс `Formatter` включает очень удобное средство, позволяющее указать аргумент, к которому должен применяться конкретный спецификатор формата. Обычно порядок аргументов и спецификаторов формата совпадает — слева направо. То есть первый спецификатор формата относится к первому аргументу, второй — ко второму аргументу и т.д. Однако, используя индекс аргумента, вы можете явно управлять тем, к какому из аргументов относится спецификатор формата.

Индекс аргумента следует сразу за символом % в спецификаторе формата. Он имеет следующий вид.

n$

Здесь *n* — индекс нужного аргумента, начиная с 1. Рассмотрим следующий пример.

```
fmt.format("%3$d %1$d %2$d", 10, 20, 30);
```

Этот код порождает такую строку.

```
30 10 20
```

В этом примере первый спецификатор формата соответствует 30, второй — 10, а третий — 20. То есть аргументы используются в порядке, отличном от простого порядка "слева направо".

Одним из преимуществ индексирования аргументов является то, что это позволяет повторно использовать аргумент, не указывая его дважды. Например, строка

```
fmt.format("%d в шестнадцатеричном формате равно %1$x", 255);
```

порождает следующий результат.

```
255 в шестнадцатеричном формате равно ff
```

Как видите, аргумент 255 используется с обоими спецификаторами формата.

Существует удобное сокращение, называемое *относительным индексом*, которое позволяет повторно использовать аргументы, соответствующие предшествующему спецификатору формата. Просто укажите < вместо индекса аргумента. Например, следующий вызов метода `format()` порождает тот же результат, что и предыдущий пример.

```
fmt.format("%d в шестнадцатеричном формате равно %<x", 255);
```

Относительные индексы особенно удобны при создании пользовательских форматов времени и даты. Рассмотрим следующий пример.

```
// Использование относительных индексов для упрощения
// создания пользовательских форматов даты и времени.
```

```
import java.util.*;

class FormatDemo6 {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();

        fmt.format("Today is day %te of %<tB, %<tY", cal);
        System.out.println(fmt);
        fmt.close();
    }
}
```

Вот как выглядит вывод.

```
Today is day 1 of January, 2011
```

Благодаря *относительной индексации, аргумент* `cal` *может быть передан только* один раз вместо трех.

Закрытие объекта класса `Formatter`

Обычно объект класса `Formatter` следует закрывать, когда завершаете его использование. Это освобождает все используемые им ресурсы, что особенно важно при форматировании вывода в файл, но и в других случаях это также может быть важно. Приведенные выше примеры демонстрируют один из способов закрыть объект класса `Formatter` — явно вызвать метод `close()`. Но начиная с JDK 7 класс `Formatter` реализует интерфейс `AutoCloseable`. Это значит, что он поддерживает новый оператор `try-с-ресурсами`. Используя этот подход, можно автоматически закрывать объект класса `Formatter`, когда он больше не нужен.

Оператор `try-с-ресурсами`, применительно к файлам, описан в главе 13, поскольку файлы — одни из наиболее часто используемых ресурсов, которые следует закрывать. Однако те же фундаментальные методики применяются и здесь. Вот первый пример применения объекта класса `Formatter`, переделанный для автоматического управления ресурсами.

```
// Использование автоматического управления ресурсами с Formatter.
import java.util.*;

class FormatDemo {
    public static void main(String args[]) {

        try (Formatter fmt = new Formatter())
        {
        fmt.format("Форматирование %s просто %d %f", " с Java", 10,
                   98.6);

            System.out.println(fmt);
        }
    }
}
```

Вывод тот же, что и прежде.

Подключение функции Java `printf()`

Хотя нет ничего технически неправильного в непосредственном применении класса `Formatter` (как это делалось в предыдущих примерах) при создании вы-

вода для отображения на консоли, существует более удобная альтернатива — метод `printf()`. Этот метод автоматически использует класс `Formatter` для создания форматированной строки. Затем он отправляет эту строку в поток `System.out`, который по умолчанию представляет собой консоль. Метод `printf()` определен и в классе `PrintStream`, и в классе `PrintWriter`. Этот метод рассматривается в главе 19.

Класс `Scanner`

Класс `Scanner` — это дополнение к классу `Formatter`. Он читает форматированный ввод и преобразует его в бинарную форму. Класс `Scanner` может применяться для чтения ввода с консоли, из файла, из строки или с другого источника, реализующего интерфейс `Readable` либо `ReadableByteChannel`. Например, вы можете использовать класс `Scanner` для чтения числа с клавиатуры и присвоения его значения переменной. Как вы увидите, несмотря на свою мощь, класс `Scanner` неожиданно прост в применении.

Конструкторы класса `Scanner`

Класс `Scanner` определяет конструкторы, перечисленные в табл. 18.16. В общем случае объект класса `Scanner` может быть создан для объекта класса `String`, `InputStream`, `File` или любого другого объекта, реализующего интерфейсы `Readable` либо `ReadableByteChannel`. Ниже приведены некоторые примеры.

Таблица 18.16. Конструкторы класса `Scanner`

Конструктор	Описание
`Scanner(File` *из*`) throws FileNotFoundException`	Создает объект класса `Scanner`, который использует указанный файл *из* в качестве входного источника
`Scanner(File` *из*`, String` *наборсимволов*`) throws FileNotFoundException`	Создает объект класса `Scanner`, который использует указанный файл *из* с кодировкой, заданной в *наборсимволов*, в качестве входного источника
`Scanner(InputStream` *из*`)`	Создает объект класса `Scanner`, который использует указанный поток *из* в качестве входного источника
`Scanner(InputStream` *из*`, String` *наборсимволов*`)`	Создает объект класса `Scanner`, который использует поток, указанный как *из* с кодировкой, заданной в *наборсимволов*, в качестве входного источника
`Scanner(Path` *из*`) throws IOException`	Создает объект класса `Sanner`, который использует файл, определенный параметром *из* как источник для ввода. (Добавлено в JDK 7)
`Scanner(Path` *из*`, String` *наборсимволов*`) throws IOException`	Создает объект класса `Sanner`, который использует файл, определенный параметром *из*, с кодировкой символов, определенной параметром *наборсимволов*, как источник для ввода. (Добавлено в JDK 7)
`Scanner(Readable` *из*`)`	Создает объект класса `Scanner`, который использует указанный параметром *из* объект интерфейса `Readable` в качестве входного источника

Окончание табл. 18.16

Конструктор	Описание
`Scanner (ReadableByteChannel из)`	Создает объект класса `Scanner`, который использует указанный параметром *из* объект интерфейса `ReadableByteChannel` в качестве входного источника
`Scanner(ReadableByteChannel из, String наборсимволов)`	Создает объект класса `Scanner`, который использует указанный параметром *из* объект интерфейса `ReadableByteChannel` с кодировкой, заданной параметром *наборсимволов*, в качестве входного источника
`Scanner(String из)`	Создает объект класса `Scanner`, который использует указанную строку в качестве входного источника

Следующая последовательность создает объект класса `Scanner`, который читает файл `Test.txt`.

```
FileReader fin = new FileReader("Test.txt");
Scanner src = new Scanner(fin);
```

Это работает, потому что класс `FileReader` реализует интерфейс `Readable`. То есть вызов конструктора интерпретируется как `Scanner(Readable)`.

Следующая строка создает объект класса `Scanner`, читающий из стандартного ввода, которым по умолчанию является клавиатура.

```
Scanner conin = new Scanner(System.in);
```

Это работает, потому что объект `System.in` — это объект класса `InputStream`. То есть вызов конструктора отображается как `Scanner(InputStream)`.

Следующая последовательность создает объект класса `Scanner`, который читает строку.

```
String instr = "10 99.88 сканирование очень просто.";
Scanner conin = new Scanner(instr);
```

Основы сканирования

После того как вы создаете объект класса `Scanner`, его очень просто использовать для чтения форматированного ввода. В общем случае объект класса `Scanner` читает *лексемы* из некоторого лежащего в основе источника, который вы указываете при создании объекта класса `Scanner`. С точки зрения класса `Scanner`, *лексема* (token) — это порция ввода, отделенная набором разделителей, которыми по умолчанию являются пробелы. Лексема читается в соответствии с определенным *регулярным выражением* (regular expression), задающим формат данных. Хотя класс `Scanner` позволяет определить специфический тип выражения, которому будет соответствовать следующая операция ввода, он включает множество предопределенных шаблонов, соответствующих элементарным типам, таким как `int` и `double`, а также строкам. Иными словами, зачастую вы не будете нуждаться в указании шаблонов.

В общем случае для использования класса `Scanner` следуйте описанной ниже процедуре.

1. Определите, доступен ли специфический тип ввода вызовом одного из методов `ScannerhasNextX`, где `X` — нужный тип данных.
2. Если ввод доступен, читайте его одним из методов `ScannernextX`.

3. Повторяйте процесс до завершения ввода.
4. Закройте объект класса `Scanner`, вызвав метод `close()`.

Как следует из сказанного выше, класс `Scanner` определяет два набора методов, которые позволяют читать ввод. Первый – это методы `hasNextX`, перечисленные в табл. 18.17. Эти методы определяют, доступен ли указанный тип ввода. Например, вызов метода `hasNextInt()` возвращает значение `true` только в том случае, если следующая лексема, подлежащая чтению, является целым числом. Если требуемые данные доступны, вы читаете их одним из методов `ScannernextX`, описанных в табл. 18.18. Например, чтобы прочесть следующее целое число, вызовите метод `nextInt()`. Приведенная ниже последовательность показывает, как читать список целых чисел, вводимых с клавиатуры.

```
Scanner conin = new Scanner(System.in);
int i;
// Читать список целых.
while(conin.hasNextInt()) {
    i = conin.nextInt();
    // ...
}
```

Таблица 18.17. Методы `hasNext` класса `Scanner`

Метод	Описание
`boolean hasNext()`	Возвращает значение `true`, если доступна для чтения следующая лексема любого типа. В противном случае возвращает значение `false`
`boolean hasNext(Pattern` *шаблон*`)`	Возвращает значение `true`, если доступна для чтения лексема, соответствующая шаблону *шаблон*. В противном случае возвращает значение `false`
`boolean hasNext(String` *шаблон*`)`	Возвращает значение `true`, если доступна для чтения лексема, соответствующая шаблону *шаблон*. В противном случае возвращает значение `false`
`boolean hasNextBigDecimal()`	Возвращает значение `true`, если доступно для чтения значение, которое можно поместить в объект `BigDecimal`. В противном случае возвращает значение `false`
`boolean hasNextBigInteger()`	Возвращает значение `true`, если доступно для чтения значение, которое можно поместить в объект `BigInteger`. В противном случае возвращает значение `false`. (По умолчанию используется основание 10)
`boolean hasNextBigInteger(int` *основание*`)`	Возвращает значение `true`, если доступно для чтения значение по основанию *основание*, которое можно поместить в объект `BigInteger`. В противном случае возвращает значение `false`
`boolean hasNextBoolean()`	Возвращает значение `true`, если доступно для чтения значение типа `boolean`. В противном случае возвращает значение `false`
`boolean hasNextByte()`	Возвращает значение `true`, если доступно для чтения значение типа `byte`. В противном случае возвращает значение `false`

Окончание табл. 18.17

Метод	Описание
`boolean hasNextByte(int` *основание*`)`	Возвращает значение `true`, если доступно для чтения значение типа `byte` с указанным основанием *основание*. В противном случае возвращает значение `false`
`boolean hasNextDouble()`	Возвращает значение `true`, если доступно для чтения значение типа `double`. В противном случае возвращает значение `false`
`boolean hasNextFloat()`	Возвращает значение `true`, если доступно для чтения значение типа `float`. В противном случае возвращает значение `false`
`boolean hasNextInt()`	Возвращает значение `true`, если доступно для чтения значение, которое можно поместить в объект `int`. В противном случае возвращает значение `false`. (По умолчанию используется основание 10)
`boolean hasNextInt(int` *основание*`)`	Возвращает значение `true`, если доступно для чтения значение типа `int` с указанным основанием *основание*. В противном случае возвращает значение `false`
`boolean hasNextLine()`	Возвращает значение `true`, если доступна строка ввода
`boolean hasNextLong()`	Возвращает значение `true`, если доступно для чтения значение типа `long`. В противном случае возвращает значение `false`. (По умолчанию используется основание 10)
`boolean hasNextLong(int long` *основание*`)`	Возвращает значение `true`, если доступно для чтения значение по основанию *основание*. В противном случае возвращает значение `false`
`boolean hasNextShort()`	Возвращает значение `true`, если доступно для чтения значение типа `short`. В противном случае возвращает значение `false`. (По умолчанию используется основание 10)
`boolean hasNextShort(int` *основание*`)`	Возвращает значение `true`, если доступно для чтения значение `short` по основанию *основание*. В противном случае возвращает значение `false`

Таблица 18.18. Методы `next` класса `Scanner`

Метод	Описание
`String next()`	Возвращает следующую лексему любого типа из входного источника
`String next(Pattern` *шаблон*`)`	Возвращает следующую лексему, соответствующую шаблону, переданному в *шаблон*, из входного источника
`String next(String` *шаблон*`)`	Возвращает следующую лексему, соответствующую шаблону, переданному в *шаблон*, из входного источника
`BigDecimal nextBigDecimal()`	Возвращает следующую лексему как объект `BigDecimal`
`BigInteger nextBigInteger()`	Возвращает следующую лексему как объект `BigInteger`. (По умолчанию используется основание 10)
`BigInteger nextBigInteger(int` *основание*`)`	Возвращает следующую лексему как объект `BigInteger`. Используется основание *основание*

Окончание табл. 18.18

Метод	Описание
`boolean nextBoolean()`	Возвращает следующую лексему как значение `boolean`
`byte nextByte()`	Возвращает следующую лексему как значение `byte`. (По умолчанию используется основание 10)
`byte nextByte(int` *основание*`)`	Возвращает следующую лексему как значение типа `byte` по основанию *основание*
`double nextDouble()`	Возвращает следующую лексему как значение типа `double`
`float nextFloat()`	Возвращает следующую лексему как значение `float`
`int nextInt()`	Возвращает следующую лексему как значение типа `int`. (По умолчанию используется основание 10)
`int nextInt(int` *основание*`)`	Возвращает следующую лексему как значение типа `int` по основанию *основание*
`String nextLine()`	Возвращает следующую строку ввода
`long nextLong()`	Возвращает следующую лексему как значение типа `long`. (По умолчанию используется основание 10)
`long nextLong(int` *основание*`)`	Возвращает следующую лексему как значение типа `long` по основанию *основание*
`short nextShort()`	Возвращает следующую лексему как значение типа `short`. (По умолчанию используется основание 10)
`short nextShort(int` *основание*`)`	Возвращает следующую лексему как значение типа `short` по основанию *основание*

В приведенном выше примере цикл `while` остановится, как только следующая лексема не будет целым числом. То есть цикл прекращает читать целые числа, как только во входном потоке обнаруживается значение, отличное от типа целого числа.

Если метод `next` не может найти тип данных, который он ожидает, передает исключение `InputMismatchException`. Исключение `NoSuchElementException` передается в случае, когда больше нет доступного ввода. Поэтому сначала лучше проверить, что данные требуемого типа доступны, с помощью метода `hasNext`, прежде чем вызывать соответствующий ему метод `next`.

Некоторые примеры применения класса `Scanner`

Класс `Scanner` способен существенно упростить ранее утомительную задачу. Давайте рассмотрим несколько примеров. Следующая программа подсчитывает среднее из списка чисел, введенных с клавиатуры.

```
// Применение Scanner для вычисления среднего из списка 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 подсчета среднего.");

        // Читать и суммировать значения.
```

```
        while(conin.hasNext()) {
            if(conin.hasNextDouble()) {
                sum += conin.nextDouble();
                count++;
            }
            else {
                String str = conin.next();
                if(str.equals("готово")) break;
                else {
                    System.out.println("Ошибка формата данных.");
                    return;
                }
            }
        }
        conin.close();
        System.out.println("Среднее равно " + sum / count);
    }
}
```

Эта программа читает числа с клавиатуры, суммирует их в процессе до тех пор, пока пользователь не введет строку "готово". Затем она прекращает ввод и отображает среднее значение введенных чисел.

Ниже показан пример ее выполнения.

```
Введите числа для подсчета среднего.
1.2
2
3.4
4
готово
Среднее равно 2.65
```

Программа читает числа до тех пор, пока не получит лексему, которую нельзя интерпретировать как корректное значение типа double. Когда подобное происходит, она проверяет, что лексема соответствует строке "готово". Если это так, программа завершается нормально. В противном случае она отображает сообщение об ошибке.

Обратите внимание на то, что числа читаются с помощью метода nextDouble(). Этот метод читает любые числа, которые могут быть преобразованы в тип double, включая целые значения вроде 2 и значения с плавающей точкой, такие как 3.4. То есть число, прочитанное методом nextDouble(), не требует наличия десятичной точки. Тот же общий принцип применим к любому методу next. Они найдут соответствие и прочитают данные в любом формате, который может представлять данные запрашиваемого типа.

Еще один заслуживающий внимания момент класса Scanner — это то, что одна и та же техника, используемая для чтения одного источника, может быть применена для другого. Например, ниже представлена измененная версия предыдущей программы, предназначенная для чтения списка чисел из файла.

```
// Использование Scanner для вычисления среднего значения числа из файла.
import java.util.*;
import java.io.*;
class AvgFile {
    public static void main(String args[]) throws IOException {
        int count = 0;
        double sum = 0.0;

        // Записать вывод в файл.
        FileWriter fout = new FileWriter("test.txt");
        fout.write("2 3.4 5 6 7.4 9.1 10.5 готово");
        fout.close();
```

```
        FileReader fin = new FileReader("Test.txt");
        Scanner src = new Scanner(fin);

        // Читать и суммировать значения.
        while(src.hasNext()) {
            if(src.hasNextDouble()) {
                sum += src.nextDouble();
                count++;
            }
            else {
                String str = src.next();
                if(str.equals("готово")) break;
                else {
                    System.out.println("Ошибка формата файла.");
                    return;
                }
            }
        }

        src.close();
        System.out.println("Среднее равно " + sum / count);
    }
}
```

Вот вывод.

```
Среднее равно 6.2
```

Приведенная выше программа иллюстрирует другое важное средство класса `Scanner`. Обратите внимание на то, что считыватель файла, упомянутый как `fin`, не закрывается непосредственно. Вместо этого он автоматически закрывается, когда объект `src` вызывает метод `close()`. Когда вы закрываете объект класса `Scanner`, связанный с ним объект интерфейса `Readable` также закрывается (если он реализует интерфейс `Closeable`). Поэтому в данном случае файл, упомянутый как `fin`, автоматически закрывается, когда закрывается объект `src`.

Начиная с JDK 7 класс `Scanner` реализует также интерфейс `AutoCloseable`. Это означает, что им может управлять блок *try-с-ресурсами*. Как описано в главе 13, когда используется оператор *try-с-ресурсами*, сканер автоматически закрывается в конце блока. Например, объектом `src` в приведенной выше программе можно управлять так.

```
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("готово")) break;
            else {
                System.out.println("Ошибка формата файла.");
                return;
            }
        }
    }
}
```

Чтобы продемонстрировать закрытие объекта класса `Scanner`, следующие примеры вызовут метод `close()` явно. Это позволяет им также компилироваться

версиями Java до JDK 7. Однако подход с оператором *try-с-ресурсами* проще и может помочь предотвратить ошибки. Его использование рекомендуется для нового кода.

Еще один момент: чтобы сохранить компактность этого и других примеров в данном разделе, исключения ввода-вывода просто передаются из метода `main()`. Однако ваш реальный код обычно будет обрабатывать исключения ввода-вывода самостоятельно.

Вы можете использовать объект класса `Scanner` для чтения ввода, который содержит некоторые различные типы данных, даже если порядок их следования заранее не известен. Вы просто должны проверять, какого типа доступны данные, прежде чем читать их. Например, рассмотрим следующую программу.

```
// Применение Scanner для чтения данных разного типа 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;

        // Писать вывод в файл.
        FileWriter fout = new FileWriter("test.txt");
        fout.write("Тестирование Scanner 10 12.2 один true два 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();
    }
}
```

Ниже показан результат.

```
String: Тестирование
String: Scanner
int: 10
double: 12.2
```

```
String: один
boolean: true
String: два
boolean: false
```

При чтении данных смешанных типов, как это делает предыдущая программа, следует быть немного внимательнее относительно порядка, в котором вызываются методы `next`. Например, если в цикле поменять порядок вызовов методов `nextInt()` и `nextDouble()`, то оба числовых значения будут прочитаны как тип `double`, поскольку метод `nextDouble()` соответствует любой строке, содержащей число, которое может быть интерпретировано как тип `double`.

Установка разделителей

Класс `Scanner` определяет, где начинаются и заканчиваются лексемы, на основе набора *разделителей*. По умолчанию разделителями являются пробельные символы, и именно этот набор разделителей используется в предыдущем примере. Однако можно изменить разделители, вызвав метод `useDelimiters()`, показанный ниже.

```
Scanner useDelimiter(String шаблон)
Scanner useDelimiter(Pattern шаблон)
```

Здесь *шаблон* — это регулярное выражение, которое определяет набор разделителей. Ниже показана измененная версия приведенной ранее программы, которая читает список чисел, разделенных запятыми и любым количеством пробелов.

```
// Применение Scanner для вычисления среднего в списке
// разделенных запятыми значений.
import java.util.*;
import java.io.*;
class SetDelimiters {
    public static void main(String args[]) throws IOException {
        int count = 0;
        double sum = 0.0;

        // Писать вывод в файл.
        FileWriter fout = new FileWriter("test.txt");

        // Теперь сохранить значения в списке, разделенном запятыми.
        fout.write("2, 3.4, 5,6, 7.4, 9.1, 10.5, готово");
        fout.close();
        FileReader fin = new FileReader("Test.txt");
        Scanner src = new Scanner(fin);

        // Установить в качестве разделителей запятые и пробелы.
        src.useDelimiter(", *");

        // Читать и суммировать значения.
        while(src.hasNext()) {
            if(src.hasNextDouble()) {
                sum += src.nextDouble();
                count++;
            }
            else {
                String str = src.next();
                if(str.equals("готово")) break;
                else {
                    System.out.println("Ошибка формата файла.");
                    return;
                }
```

```
            }
        }
        src.close();
        System.out.println("Среднее равно " + sum / count);
    }
}
```

В этой версии числа, записанные в файл `test.txt`, разделены запятыми и пробелами. Применение шаблона разделителей `",*"` сообщает объекту класса `Scanner`, чтобы он воспринимал запятую и ноль в качестве разделителей. Вывод программы будет таким же, что и ранее.

Вы можете получить текущий шаблон разделителей вызовом метода `delimiter()`, показанного ниже.

```
Pattern delimiter()
```

Прочие возможности класса `Scanner`

Класс `Scanner` определяет несколько других методов в дополнение к уже упомянутым. В частности, одним из наиболее часто используемых в некоторых случаях является метод `findInLine()`. Его общие формы представлены ниже.

```
String findInLine(Pattern шаблон)
String findInLine(String шаблон)
```

Этот метод ищет указанный шаблон внутри следующей строки текста. Если шаблон найден, соответствующая ему лексема принимается и возвращается. В противном случае возвращается значение `null`. Метод работает независимо от набора разделителей. Этот метод удобен, если требуется специфический шаблон. Например, следующая программа ищет поле возраста во входной строке и затем отображает его.

```
// Демонстрация применения findInLine().
import java.util.*;
class FindInLineDemo {
    public static void main(String args[]) {
        String instr = "Имя: Том Возраст: 28 ID: 77";
        Scanner conin = new Scanner(instr);

        // Найти и отобразить возраст.
        conin.findInLine("Возраст:"); // найти "Возраст"
        if(conin.hasNext())
            System.out.println(conin.next());
        else
            System.out.println("Ошибка!");
        conin.close();
    }
}
```

Результатом будет `28`. В программе метод `findInLine()` применяется для поиска вхождения шаблона `"Возраст:"`. Когда он найден, читается следующая лексема, которая представляет собой значение возраста.

С методом `findInLine()` связан метод `findWithinHorixon()`.

```
String findWithinHorizon(Pattern шаблон, int количество)
String findWithinHorizon(String шаблон, int количество)
```

Этот метод пытается найти вхождение указанного шаблона в следующие *количество* символов. В случае успеха метод возвращает совпадающий шаблон, в противном случае — значение `null`. Если параметр *количество* содержит значение нуль, то поиск выполняется во всем вводе до тех пор, пока либо не будет обнаружено совпадение, либо не будет достигнут конец входной информации.

Вы можете пропустить шаблон, используя метод `skip()`.

```
Scanner skip(Pattern шаблон)
Scanner skip(String шаблон)
```

Если соответствие шаблону имеется, метод `skip()` просто пропускает его и возвращает ссылку на вызывающий объект. Если шаблон не найден, метод `skip()` передает исключение `NoSuchElementException`.

К другим методам класса `Scanner` относятся метод `radix()`, который возвращает основание системы счисления по умолчанию, используемое классом `Scanner` при чтении чисел; метод `useRadix()`, который устанавливает основание системы счисления; метод `reset()`, который сбрасывает сканер; и метод `close()`, закрывающий сканер.

Классы `ResourceBundle`, `ListResourceBundle` и `PropertyResourceBundle`

Пакет `java.util` включает три класса, предназначенные для интернационализации ваших программ. Первый из них — абстрактный класс `ResourceBundle`. Он определяет методы, позволяющие управлять коллекцией чувствительных к региональным данным ресурсов, таких как строки, используемые в качестве меток элементов пользовательского интерфейса программ. Вы можете определить два или более наборов переведенных строк, поддерживающих различные языки, скажем, английский, немецкий или китайский, причем каждый перевод будет находиться в собственной связке (bundle). Затем вы можете загружать нужную связку в соответствии с текущими локальными установками и использовать строки для создания пользовательского интерфейса программы.

Связки ресурсов идентифицируются *именем семейства* (также называемыми их *базовыми именами*). К имени семейства может быть добавлен двухсимвольный *код языка*, указывающий на конкретный язык. В этом случае, если запрошенные региональные данные соответствуют коду языка, используется эта версия связки ресурсов. Например, связка ресурсов с именем семейства `SampleRB` может иметь немецкую версию `SampleRB_de` и русскую версию `SampleRB_ru`. (Обратите внимание: знак подчеркивания связывает имя семейства с кодом языка.) Таким образом, если текущими региональными данными есть `Locale.GERMAN`, будет применяться связка `SampleRB_de`.

Также можно указать специфические варианты языка, которые относятся к определенной стране, указывая *код страны* после кода языка. Код страны — это двухсимвольный идентификатор в верхнем регистре, такой как `AU` для Австрии или `IN` для Индии. Коду страны также предшествует знак подчеркивания, когда он связывается с именем связки ресурсов. Связка ресурсов, имеющая только имя семейства, применяется по умолчанию. Она используется, когда недоступны специфичные для языка связки.

На заметку! Коды языков определены стандартом ISO 639, а коды стран — стандартом ISO 3166.

Методы, определенные в классе `ResourceBundle`, перечислены в табл. 18.19. Одно важное замечание: пустые ключи не допускаются, а потому несколько методов передают исключение `NullPointerException`, если получают пустой ключ. Следует обратить внимание на вложенный класс `ResourceBundle.Control`. Он добавлен в Java SE 6 и используется для управления процессом загрузки связок ресурсов.

Таблица 18.19. Методы, определенные в классе ResourceBundle

Метод	Описание
`static final void clearCache()`	Удаляет все связки ресурсов из кеша, которые были загружены загрузчиком классов по умолчанию
`static final void clearCache(ClassLoader загр)`	Удаляет все связки ресурсов из кеша, которые были загружены загрузчиком *загр*
`boolean containsKey(String k)`	Возвращает значение `true`, если *k* — ключ внутри вызывающей связки ресурсов (или ее родителя)
`static final ResourceBundle getBundle(String имяСемейства)`	Загружает связку ресурсов с именем семейства *имяСемейства*, используя региональные данные и загрузчик классов по умолчанию. Передает исключение `MissingResourceException`, если не найдено связки ресурсов, соответствующей имени *имяСемейства*
`static final ResourceBundle getBundle(String имяСемейства, Locale регион)`	Загружает связку ресурсов с именем семейства *имяСемейства*, используя указанные региональные данные и загрузчик классов по умолчанию. Передает исключение `MissingResourceException`, если не найдено связки ресурсов, соответствующей имени *имяСемейства*
`static ResourceBundle getBundle(String имяСемейства, Locale регион, ClassLoader загр)`	Загружает связку ресурсов с именем семейства *имяСемейства*, используя указанные региональные данные и загрузчик классов. Передает исключение `MissingResourceException`, если не найдено связки ресурсов, соответствующей имени *имяСемейства*
`static final ResourceBundle getBundle(String имяСемейства, ResourceBundle.Control контроль)`	Загружает связку ресурсов с именем семейства *имяСемейства*, используя региональные данные и загрузчик классов по умолчанию. Процесс загрузки находится под управлением *контроль*. Передает исключение `MissingResourceException`, если не найдено связки ресурсов, соответствующей имени *имяСемейства*
`static final ResourceBundle getBundle(String имяСемейства, Locale регион, ResourceBundle.Control контроль)`	Загружает связку ресурсов с именем семейства *имяСемейства*, используя указанные региональные данные и загрузчик классов по умолчанию. Процесс загрузки находится под управлением *контроль*. Передает исключение `MissingResourceException`, если не найдено связки ресурсов, соответствующей имени *имяСемейства*
`static ResourceBundle getBundle(String имяСемейства, Locale регион, ClassLoader загр, ResourceBundle.Control контроль)`	Загружает связку ресурсов с именем семейства *имяСемейства*, используя указанные региональные данные и загрузчик классов. Процесс загрузки находится под управлением *контроль*. Передает исключение `MissingResourceException`, если не найдено связки ресурсов, соответствующей имени *имяСемейства*
`abstract Enumeration<String>getKeys()`	Возвращает ключи связки ресурсов как перечисление строк. Любые родительские ключи также возвращаются
`Locale getLocale()`	Возвращает региональные данные, поддерживаемые связкой ресурсов

Окончание табл. 18.19

Метод	Описание
`final Object getObject(String k)`	Возвращает объект, ассоциированный с ключом, переданным параметром *k*. Передает исключение `MissingResourceException`, если *k* не найден в связке ресурсов
`final String getString(String k)`	Возвращает строку, ассоциированную с ключом, переданным параметром *k*. Передает исключение `MissingResourceException`, если *k* не найден в связке ресурсов. Передает исключение `ClassCastException`, если объект, ассоциированный с *k*, не является строкой
`final String[] getStringArray(String k)`	Возвращает массив строк, ассоциированных с ключом, переданным параметром *k*. Передает исключение `MissingResourceException`, если *k* не найден в связке ресурсов. Передает исключение `MissingResourceException`, если объект, ассоциированный с *k*, не является массивом строк
`protected abstract Object handleGetObject(String k)`	Возвращает объект, ассоциированный с ключом, переданным параметром *k*. Возвращает значение `null`, если *k* не найден в связке ресурсов
`protected Set<String> handleKeySet()`	Возвращает ключи связки ресурсов как набор. Родительские ключи не извлекаются. Также не возвращаются ключи со значениями `null`
`Set<String> keySet()`	Возвращает ключи связки ресурсов как набор строк. Родительские ключи также извлекаются
`protected void setParent(ResourceBundle` *родительский*`)`	Устанавливает *родительский* как родительскую связку для данной связки ресурсов. Если ключ не найден в вызывающем ресурсном объекте, поиск продолжится в родительском объекте

У класса `ResourceBundle` есть два подкласса. Первый из них — класс `PropertyResourceBundle`, управляющий ресурсами через файлы свойств. Этот класс не добавляет собственных методов.

Второй — это абстрактный класс `ListResourceBundle`, управляющий ресурсами в массиве пар "ключ-значение". Этот класс добавляет метод `getContents()`, который должны реализовать все подклассы. Выглядит он так.

```
protected abstract Object[][] getContents()
```

Этот класс возвращает двухмерный массив, содержащий пары "ключ-значение", представляющие ресурсы. Ключи могут быть строками. Значения — обычно строки, но могут быть и объектами других типов.

Рассмотрим пример, демонстрирующий использование связки ресурсов. Связка ресурсов имеет имя семейства `SampleRB`. Два класса связок ресурсов этого семейства создаются расширением класса `ListResourceBundle`. Первый называется `SampleRB` и представляет собой связку по умолчанию для английского языка.

```
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;
    }
}
```

Вторая связка ресурсов, показанная ниже, называется `SampleRB_de` и содержит немецкий перевод.

```
import java.util.*;

// Немецкоязычная версия.
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;
    }
}
```

Следующая программа демонстрирует эти две связки ресурсов, отображая строки, ассоциированные с каждым ключом как для версии по умолчанию (английской), так и для немецкой версии.

```
// Демонстрация связки ресурсов.
import java.util.*;

class LRBDemo {
    public static void main(String args[]) {
        // Загрузить связку по умолчанию.
        ResourceBundle rd = ResourceBundle.getBundle("SampleRB");

        System.out.println("Англоязычная 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 немецкую связку.
        rd = ResourceBundle.getBundle("SampleRB", Locale.GERMAN);

        System.out.println("\nНемецкоязычная 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"));
    }
}
```

Вывод этой программы будет таким.

```
Англоязычная версия:
Строка для ключа Title: My Program
Строка для ключа StopText: Stop
Строка для ключа StartText: Start

Немецкоязычная версия:
Строка для ключа Title: Mein Programm
Строка для ключа StopText: Anschlag
Строка для ключа StartText: Anfang
```

Прочие служебные классы и интерфейсы

В дополнение к уже описанным классам, пакет `java.util` содержит классы, перечисленные в табл. 18.20.

Таблица 18.20. Дополнительные классы пакета `java.util`

Класс	Описание
`EventListenerProxy`	Расширяет класс `EventListener` для принятия дополнительных параметров. См. в главе 23 обсуждение слушателей событий
`EventObject`	Суперкласс для всех классов событий. События обсуждаются в главе 23
`FormattableFlags`	Определяет флаги форматирования, используемые в интерфейсе `Formattable`
`Objects`	Различные методы, которые работают с объектами. (Добавлено в JDK 7)
`PropertyPermission`	Управляет правами доступа к свойствам
`ServiceLoader`	Обеспечивает средства нахождения поставщиков служб
`UUID`	Инкапсулирует *универсальные уникальные идентификаторы* (Universally Unique Identifier – UUID) и управляет ими

Интерфейсы, описанные в табл. 18.21, также входят в состав пакета `java.util`.

Таблица 18.21. Дополнительные интерфейсы пакета `java.util`

Интерфейс	Описание
`EventListener`	Указывает, что класс является слушателем событий. События обсуждаются в главе 23
`Formattable`	Описывает класс, обеспечивающий специальное (настраиваемое) форматирование

Вложенные пакеты `java.util`

Java определяет следующие вложенные пакеты `java.util`.

- `java.util.concurrent`
- `java.util.concurrent.atomic`
- `java.util.concurrent.locks`
- `java.util.jar`
- `java.util.logging`

- java.util.prefs
- java.util.regex
- java.util.spi
- java.util.zip

Ниже все они кратко описаны.

Пакеты java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks

Пакет java.util.concurrent, наряду с его двумя внутренними пакетами, java.util.concurrent.atomic и java.util.concurrent.locks, предназначен для поддержки параллельного программирования. Эти пакеты предлагают высокопроизводительную альтернативу применению встроенных в Java средств синхронизации, когда требуются безопасные в отношении потоков операции. Начиная с JDK 7 пакет java.util.concurrent также поддерживает инфраструктуру Fork/Join Framework. Эти пакеты подробно рассматриваются в главе 27.

Пакет java.util.jar

Этот пакет предлагает возможность чтения и записи архивных файлов Java Archive (JAR).

Пакет java.util.logging

Данный пакет обеспечивает поддержку журналов активности программ, которые могут быть использованы для записи действий программ, а также для поиска проблем и отладки.

Пакет java.util.prefs

Этот пакет обеспечивает поддержку пользовательских предпочтений. Обычно применяется для поддержки конфигураций программ.

Пакет java.util.regex

Этот пакет обеспечивает поддержку работы с регулярными выражениями. Он подробно описан в главе 27.

Пакет java.util.spi

Этот пакет обеспечивает поддержку поставщиков служб.

Пакет java.util.zip

Данный пакет обеспечивает возможность чтения и записи файлов архивов в популярных форматах ZIP и GZIP. Доступны также входные и выходные потоки ZIP и GZIP.

ГЛАВА 19

Ввод-вывод: пакет java.io

Эта глава посвящена пакету `java.io`, поддерживающему операции ввода-вывода. В главе 13 представлен краткий обзор системы ввода-вывода Java, включая базовые методики чтения и записи файлов, обработки исключений ввода-вывода и закрытия файла. Здесь же рассмотрим систему ввода-вывода Java более подробно.

Как известно всем программистам с давних времен, большинство программ не может выполнять свою работу, не имея доступа к внешним данным. Данные извлекаются из источника *ввода*. Результат программы направляется в *вывод*. На языке Java эти понятия определяются очень широко. Например, источником ввода или местом вывода может служить сетевое соединение, буфер памяти или дисковый файл — всеми ими можно манипулировать при помощи классов ввода-вывода Java. Хотя физически они совершенно различны, все эти устройства описываются единой абстракцией — *потоком*. Поток, как уже объяснялось в главе 13, — это логическая сущность, которая выдает или получает информацию. Поток присоединен к физическому устройству при помощи системы ввода-вывода Java. Все потоки ведут себя похоже, даже несмотря на то, что физические устройства, к которым они присоединены, в корне отличаются.

На заметку! Потоковая система ввода-вывода, используемая пакетом `java.io` и описанная в этой главе, была частью языка Java начиная с его первого выпуска и широко используется до сих пор. Однако начиная с версии 1.4 в язык Java была добавлена вторая система ввода-вывода. Она называется NIO (что первоначально было акронимом от New I/O (новый ввод-вывод)). Система NIO расположена в пакете `java.nio` и его внутренних пакетах. С выпуском комплекта JDK 7 возможности системы NIO были существенно расширены, и популярность ее использования, как ожидается, возрастет. Система NIO описана в главе 20.

Классы и интерфейсы ввода-вывода Java

Классы ввода-вывода, определенные в пакете `java.io`, перечислены ниже.

`BufferedInputStream`	`FileWriter`	`PipedOutputStream`
`BufferedOutputStream`	`FilterInputStream`	`PipedReader`
`BufferedReader`	`FilterOutputStream`	`PipedWriter`
`BufferedWriter`	`FilterReader`	`PrintStream`
`ByteArrayInputStream`	`FilterWriter`	`PrintWriter`
`ByteArrayOutputStream`	`InputStream`	`PushbackInputStream`

CharArrayReader	InputStreamReader	PushbackReader
CharArrayWriter	LineNumberReader	RandomAccessFile
Console	ObjectInputStream	Reader
DataInputStream	ObjectInputStream.GetField	SequenceInputStream
DataOutputStream	ObjectOutputStream	SerializablePermission
File	ObjectOutputStream.PutField	StreamTokenizer
FileDescriptor	ObjectStreamClass	StringReader
FileInputStream	ObjectStreamField	StringWriter
FileOutputStream	OutputStream	Writer
FilePermission	OutputStreamWriter	
FileReader	PipedInputStream	

Пакет `java.io` также содержит два устаревших (deprecated) класса, которые не показаны в приведенном выше перечне, а именно — классы `LineNumberInputStream` и `StringBufferInputStream`. Эти классы не должны использоваться в новом коде.

В пакете `java.io` определены следующие интерфейсы.

Closeable	FileFilter	ObjectInputValidation
DataInput	FilenameFilter	ObjectOutput
DataOutput	Flushable	ObjectStreamConstants
Externalizable	ObjectInput	Serializable

Как видите, в пакете `java.io` присутствует множество классов и интерфейсов. Среди них — байтовые и символьные потоки, сериализация объектов (их сохранение и восстановление). В настоящей главе рассматривается несколько наиболее широко используемых компонентов ввода-вывода. Новый класс `Console` также описан. Начнем обсуждение с одного из наиболее отличающихся классов ввода-вывода — `File`.

Класс `File`

Хотя большинство классов, определенных в пакете `java.io`, работают с потоками, класс `File` этого не делает. Он имеет дело непосредственно с файлами и файловой системой. То есть класс `File` не указывает, как извлекается и сохраняется информация в файлах; он описывает свойства самих файлов. Объект класса `File` служит для получения информации и манипулирования информацией, ассоциированной с дисковым файлом, такой как права доступа, время, дата и путь к каталогу, а также для навигации по иерархиям подкаталогов.

На заметку! Интерфейс `Path` и класс `Files`, добавленные в систему NIO комплектом JDK 7, являются серьезной альтернативой классу `File` во многих случаях. Более подробная информация по этой теме приведена в главе 20.

Класс `File` — первичный источник и место назначения для данных во многих программах. Хотя существует несколько ограничений в части использования файлов в аплетах (из соображений безопасности), тем не менее они продолжают оставаться центральным ресурсом для хранения постоянной и разделяемой информации.

Каталог в Java трактуется как объект класса `File` с единственным дополнительным свойством – списком имен файлов, которые могут быть получены методом `list()`.

Для создания объектов класса `File` могут быть использованы следующие конструкторы.

```
File(String путьКкаталогу)
File(String путьКкаталогу, String имяфайла)
File(File объектКаталога, String имяфайла)
File(URI объектURI)
```

Здесь *путьКкаталогу* – это путь к файлу; *имяфайла* – имя файла или подкаталога; *объектКаталога* – объект класса `File`, указывающий каталог; а *объектURI* – объект `URI`, описывающий файл.

В следующем примере создается три файла: `f1`, `f2` и `f3`. Первый объект класса `File` создается с путем к каталогу в единственном аргументе. Второй включает два аргумента – путь и имя файла. Третий включает путь, присвоенный файлу `f1`, и имя файла; объект 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! Java корректно обращается с разделителями пути, которые отличаются в UNIX и Windows. Если вы используете прямой слеш (`/`) в Windows-версии Java, то путь будет все равно сформирован корректно. Помните, однако, что для использования символа обратного слеша (`\`) в Windows следует применять в строках управляющую последовательность (`\\`).

Класс `File` определяет множество методов, представляющих стандартные свойства объекта класса `File`. Например, метод `getName()` возвращает имя файла, метод `getParent()` – имя родительского каталога, а метод `exists()` – значение `true`, если файл существует, и значение `false` – если нет. Однако класс `File` не симметричен. В нем есть несколько методов, позволяющих *проверять* свойства простого файлового объекта, у которых нет дополняющих их методов изменения этих атрибутов.

В следующем примере демонстрируется применение нескольких методов класса `File`. Здесь подразумевается, что в корневом каталоге существует каталог по имени `java` и что он содержит файл по имени `COPYRIGHT`.

```
// Демонстрация работы с 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() ? "доступен для записи" :
                          "не доступен для записи");
        p(f1.canRead() ? "доступен для чтения" :
                         "не доступен для чтения");
        p(f1.isDirectory() ? "является каталогом" :
                             "не является каталогом");
        p(f1.isFile() ? "является обычным файлом" :
```

```
                    "может быть именованным каналом");
        p(f1.isAbsolute() ? "является абсолютным" :
                            "не является абсолютным");
        p("Время модификации: " + f1.lastModified());
        p("Размер: " + f1.length() + " байт");
    }
}
```

Эта программа выдает нечто вроде следующего.

```
Имя файла: COPYRIGHT
Путь: \java\COPYRIGHT
Абсолютный путь: \java\COPYRIGHT
Родительский каталог: \java
существует
доступен для записи
доступен для чтения
не является каталогом
является обычным файлом
не является абсолютным
Время модификации: 1282832030047
Размер: 695 байт
```

Большинство методов класса `File` самоочевидно, но методы `isFile()` и `isAbsolute()` — нет. Метод `isFile()` возвращает значение `true`, если вызывается с файлом, и значение `false` — если с каталогом. Также метод `isFile()` возвращает значение `false` для некоторых специальных файлов, таких как драйверы устройств и именованные каналы, поэтому этот метод может применяться для гарантии того, что данный файл действительно ведет себя как файл. Метод `isAbsolute()` возвращает значение `true`, если файл имеет абсолютный путь, и значение `false` — если относительный.

Класс `File` включает также два полезных служебных метода. Первый из них, метод `renameTo()`, показан ниже:

```
boolean renameTo(File новоеИмя)
```

Здесь имя файла, указанное в параметре *новоеИмя*, становится новым именем вызывающего объекта класса `File`. Метод возвращает значение `true` в случае успеха и значение `false` — в случае неудачи, если файл не может быть переименован (если вы пытаетесь переименовать файл, указывая имя существующего файла).

Второй служебный метод — `delete()`, который удаляет дисковый файл, представленный путем вызывающего объекта класса `File`.

```
boolean delete()
```

Также вы можете использовать метод `delete()` для удаления каталога, если он пуст. Метод `delete()` возвращает значение `true`, если ему удается удалить файл, и значение `false`, если файл не может быть удален. В табл. 19.1 приведено еще несколько методов класса `File`, которые вы наверняка сочтете полезными.

Таблица 19.1. Полезные методы класса `File`

Метод	Описание
`void deleteOnExit()`	Удаляет файл, ассоциированный с вызывающим объектом, по завершении работы виртуальной машины Java
`long getFreeSpace()`	Возвращает количество свободных байтов хранилища, доступных в разделе, ассоциированном с вызывающим объектом
`long getTotalSpace()`	Возвращает емкость хранилища раздела, ассоциированного с вызывающим объектом

Окончание табл. 19.1

Метод	Описание
`long getUsableSpace()`	Возвращает количество доступных, годных к употреблению свободных байтов, находящихся в разделе, ассоциированном с вызывающим объектом
`boolean isHidden()`	Возвращает значение `true`, если вызывающий файл является скрытым, и значение `false` – в противном случае
`boolean setLastModified(long` *миллисекунд*`)`	Устанавливает временную метку для вызываемого файла в значение *миллисекунд*, которое представляет количество миллисекунд, прошедших с 1 января 1970 г., по UTC
`boolean setReadOnly()`	Делает вызывающий файл доступным только для чтения

Также существуют методы, помечающие файлы как доступные только для чтения, записи или выполнения. Поскольку класс `File` реализует интерфейс `Comparable`, также поддерживается метод `compareTo()`.

Комплект JDK 7 добавляет в класс `File` новый метод по имени `toPath()`, который выглядит так.

```
Path toPath()
```

Метод `toPath()` возвращает объект интерфейса `Path`, который представляет файл, инкапсулируемый вызываемым объектом класса `File`. (Другими словами, метод `toPath()` преобразует объект класса `File` в объект интерфейса `Path`.) `Path` – это новый интерфейс, добавленный в JDK 7. Он находится в пакете `java.nio.file` и является частью системы NIO. Таким образом, метод `toPath()` формирует мост между старым классом `File` и новым интерфейсом `Path`. (Более подробная информация об интерфейсе `Path` приведена в главе 20.)

Каталоги

Каталог – это объект класса `File`, содержащий список других файлов и каталогов. После создания объекта класса `File`, являющегося каталогом, его метод `isDirectory()` вернет значение `true`. В этом случае вы можете вызвать для этого объекта метод `list()`, чтобы извлечь список других файлов и каталогов, находящихся внутри него. Упомянутый метод имеет две формы. Вот первая из них.

```
String[] list()
```

Список файлов возвращается в виде массива объектов класса `String`. Приведенная ниже программа иллюстрирует использование метода `list()` для просмотра содержимого каталога.

```
// Использование каталогов.
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] + " является каталогом");
```

```
                } else {
                    System.out.println(s[i] + " является файлом");
                }
            }
        } else {
            System.out.println(dirname + " is not a directory");
        }
    }
}
```

Ниже приведен вывод этой программы. (Конечно, вывод, который вы увидите, может отличаться, — в зависимости от того, что находится в каталоге.)

```
Каталог /java
bin является каталогом
lib является каталогом
demo является каталогом
COPYRIGHT является файлом
README является файлом
index.html является файлом
include является каталогом
src.zip является файлом
src является каталогом
```

Использование интерфейса `FilenameFilter`

Нередко необходимо ограничить количество файлов, возвращенных методом `list()`, для включения только тех файлов, которые соответствуют определенному шаблону имен, или *фильтру*. Для этого следует использовать вторую форму метода `list()`.

```
String[] list(FilenameFilter объектFF)
```

В этой форме *объектFF* — это объект класса, реализующего интерфейс `FilenameFilter`. Интерфейс `FilenameFilter` определяет единственный метод `accept()`, вызываемый по одному разу с каждым файлом в списке. Его общая форма такова.

```
boolean accept(File каталог, String имяфайла)
```

Метод `accept()` возвращает значение `true` для файлов каталога, указанного в *каталог*, которые должны быть включены в список (т.е. тех, что соответствуют аргументу *имяфайла*), и значение `false` — для файлов, которые следует из списка исключить.

Класс `OnlyExt`, показанный ниже, реализует интерфейс `FilenameFilter`. Он будет использован для модификации предыдущей программы, чтобы ограничить видимость имен файлов, возвращенных методом `list()`, только теми из них, которые оканчиваются расширением, указанным при создании этого 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);
    }
}
```

Модифицированная программа просмотра списка каталога показана ниже. Теперь она выведет только файлы с расширением .html.

```
// Каталог файлов .HTML.
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]);
        }
    }
}
```

Альтернатива — метод listFiles()

Существует вариация метода list(), именуемая listFiles(), которую вы можете счесть удобной. Сигнатуры метода listFiles() показаны ниже.

```
File[ ] listFiles( )
File[ ] listFiles(FilenameFilter объектFF)
File[ ] listFiles(FileFilter объектF)
```

Эти методы возвращают список файлов в виде массива объектов класса File вместо строк. Первый метод возвращает все файлы, второй — только те, что удовлетворяют указанному интерфейсом FilenameFilter. Помимо возвращения массива объектов класса File, эти две версии метода listFiles() работают точно так же, как и методы list().

Третья версия метода listFiles() возвращает те файлы, путевые имена которых соответствуют указанному объектом интерфейса FileFilter. Интерфейс FileFilter определяет единственный метод accept(), который вызывается один раз для каждого файла в списке. Его общая форма такова.

```
boolean accept(File путь)
```

Метод accept() возвращает значение true для файлов, которые должны быть включены в список (т.е. тех, что соответствуют аргументу *путь*), и значение false — для тех, которые следует исключить.

Создание каталогов

Еще двумя полезными служебными методами класса File являются mkdir() и mkdirs(). Метод mkdir() создает каталог, возвращая значение true в случае успеха и значение false — в случае неудачи. Неудача может произойти по разным причинам, например путь, указанный в объекте класса File, уже существует или каталог не может быть создан по причине того, что полный путь к нему еще не существует. Чтобы создать каталог, для которого путь еще не создан, используйте метод mkdirs(). Он создаст как сам каталог, так и всех его родителей.

Интерфейсы AutoCloseable, Closeable и Flushable

Для потоковых классов весьма важны три интерфейса. Два из них — интерфейсы Closeable и Flushable — были определены в пакете java.io и добавлены в комплекте JDK 5. Третий интерфейс — AutoCloseable — был добавлен в комплекте JDK 7 и расположен в пакете java.lang.

Интерфейс AutoCloseable осуществляет в комплекте JDK 7 поддержку нового оператора try-*с-ресурсами*, который автоматизирует процесс закрытия ресурса (см. главу 13). Только объекты классов, реализующих интерфейс AutoCloseable, могут управляться оператором try-*с-ресурсами*. Интерфейс AutoCloseable обсуждается в главе 16, но здесь, для удобства, приведен его краткий обзор. Интерфейс AutoCloseable определяет только метод close().

```
void close() throws Exception
```

Этот метод закрывает вызывающий объект, высвобождая любые ресурсы, которые он может использовать. Метод вызывается автоматически в конце оператора try-*с-ресурсами*, избавляя таким образом от необходимости явно вызывать метод close(). Поскольку этот интерфейс реализуется всеми классами ввода-вывода, которые открывают поток, такие потоки могут быть автоматически закрыты оператором try-*с-ресурсами*. Автоматическое закрытие потоков гарантирует правильность их закрытия, когда они больше не нужны, предотвращая таким образом утечку памяти и другие проблемы.

Интерфейс Closeable также определяет метод close(). Объекты этого класса, реализующего интерфейс Closeable, могут быть закрыты. Начиная с комплекта JDK 7 интерфейс Closeable расширяет интерфейс AutoCloseable. Поэтому в комплекте JDK 7 любой класс, который реализует интерфейс Closeable также, реализует интерфейс AutoCloseable.

Объекты класса, реализующего интерфейс Flushable, могут заставить буферизованный вывод записываться в поток, к которому присоединен данный объект. Он определяет метод flush(), показанный ниже.

```
void flush() throws IOException
```

Сброс потока обычно вынуждает буферизованный вывод физически записываться на лежащем в основе устройстве. Этот интерфейс реализован всеми классами ввода-вывода, способными выполнять запись в поток.

Исключения ввода-вывода

Два исключения играют важную роль в обработке ввода-вывода. Первое из них — исключение IOException — имеет отношение к большинству классов ввода-вывода, описанных в данной главе, поэтому при ошибке ввода-вывода происходит передача исключения IOException. В большинстве случаев, если файл не может быть открыт, передается исключение FileNotFoundException. Класс исключения FileNotFoundException происходит от класса IOException, поэтому оба могут быть обработаны в одном блоке catch, предназначенном для обработки исключения IOException. Этот подход используется для краткости в большинстве примеров кода данной главы. Однако в собственных приложениях вам может иметь смысл обрабатывать их по отдельности.

Другой класс исключения, который иногда очень важен при выполнении ввода-вывода, — это класс SecurityException. Как описано в главе 13, когда присут-

ствует менеджер безопасности, некоторые классы файлов передают исключение SecurityException при попытке открыть файл с нарушением безопасности. По умолчанию приложения, запущенные при помощи команды java, не используют менеджер безопасности. Поэтому примеры ввода-вывода в этой книге не отслеживают возможность передачи исключения SecurityException. Однако аплеты будут использовать менеджер безопасности, предоставленный браузером, и файловый ввод-вывод, выполняемый аплетом, может передать исключение SecurityException. В таком случае вам придется обрабатывать и это исключение.

Два способа закрытия потока

Как правило, поток следует закрыть, когда он больше не нужен. Если не сделать этого, может произойти утечка памяти и потеря ресурсов. Методики закрытия потока были описаны в главе 13, но из-за важности кратко напомню их здесь, прежде чем перейти к рассмотрению потоковых классов.

Начиная с JDK 7 существует два основных способа, которыми вы можете закрыть поток. Первый подразумевает явный вызов метода close() для потока. Это традиционный подход, который использовался начиная с первого выпуска Java. При этом подходе метод close() обычно вызывается в блоке finally. Таким образом, упрощенный шаблон традиционного подхода выглядит так.

```
try {
    // открыть и использовать файл
} catch(исключениеВвода-вывода) {
    // ...
} finally {
    // закрыть файл
}
```

Эта общая методика (или его разновидность) популярна в коде, предшествующем JDK 7.

Второй подход закрытия потока подразумевает автоматизацию процесса с использованием нового оператора try-с-ресурсами, который появился в комплекте JDK 7. Оператор try-с-ресурсами — это улучшенная форма оператора try, имеющего следующую форму.

```
try (спецификация-ресурса) {
    // использование ресурса
}
```

Здесь *спецификация-ресурса* — это оператор или операторы, которые объявляют и инициализируют ресурс, такой как файл или другой связанный с потоком ресурс. Он состоит из объявления переменной, в котором переменная инициализируется ссылкой на управляемый объект. Когда заканчивается блок try, ресурс освобождается автоматически. В случае файла это означает, что он автоматически закрывается. Таким образом, нет никакой необходимости вызывать метод close() явно.

Вот три ключевых пункта, касающихся оператора try-с-ресурсами.

- Ресурсы, управляемые оператором try-с-ресурсами, должны быть объектами классов, реализующих интерфейс AutoCloseable.
- Ресурс, объявленный в блоке try, является неявно финальным.
- Вы можете управлять несколькими ресурсами, отделив каждый из них в объявлении точкой с запятой.

Кроме того, не забывайте, что область видимости объявленного ресурса ограничивается оператором `try-с-ресурсами`.

Основное преимущество оператора `try-с-ресурсами` заключается в том, что ресурс (в данном случае — поток) закрывается автоматически по завершении блока `try`. Таким образом, невозможно забыть закрыть поток, например. Кроме того, подход с оператором `try-с-ресурсами` дает обычно более краткий и понятный исходный код, который проще поддерживать.

Благодаря своим преимуществам, оператор `try-с-ресурсами`, как ожидается, будет интенсивно использоваться в новом коде. В результате большая часть кода этой главы (и книги) будет использовать его. Однако поскольку существуют миллионы строк кода, написанного до появления комплекта JDK 7, программисты должны быть знакомы с традиционным подходом закрытия потока. Например, вам, весьма вероятно, придется работать с устаревшим кодом, который применяет традиционный подход, или в среде, которая использует версию Java, предшествующую комплекту JDK 7. Может сложиться ситуация, когда автоматизированный подход окажется неприменим из-за других аспектов вашего кода. Поэтому несколько примеров ввода-вывода в этой книге демонстрируют традиционный подход, и вы сможете увидеть его в действии.

Еще один момент: примеры, которые используют оператор `try-с-ресурсами`, следует компилировать в комплекте JDK 7 или позже. Они не будут работать со старым компилятором. Примеры, которые используют традиционный подход, могут быть откомпилированы прежними версиями Java.

Помните! Поскольку оператор `try-с-ресурсами` упрощает процесс освобождения ресурсов и устраняет возможность их случайного пропуска, этот подход рекомендуется для нового кода, когда его использование возможно.

Классы потоков

Основанный на потоках ввода-вывода, Java построен на базе абстрактных классов: `InputStream`, `OutputStream`, `Reader` и `Writer`. Эти классы уже были кратко упомянуты в главе 13. Они используются для создания некоторых конкретных подклассов потоков.

Хотя ваши программы реализуют свои операции ввода-вывода через конкретные подклассы, классы верхнего уровня определяют базовые функциональные возможности, общие для всех потоковых классов.

Классы `InputStream` и `OutputStream` предназначены для байтовых потоков, а абстрактные классы `Reader` и `Writer` — для символьных. Классы байтовых и символьных потоков формируют отдельные иерархии. В целом классы символьных потоков следует использовать, имея дело с символами строк, а классы байтовых потоков — работая с байтами или другими двоичными объектами.

Далее в этой главе мы будем рассматривать как байтовые, так и символьные потоки.

Байтовые потоки

Классы байтовых потоков предоставляют богатую среду для обработки байтового ввода-вывода. Байтовый поток может быть использован с объектами любого типа, включая двоичные данные. Такая многосторонность делает байтовые потоки

важными для многих типов программ. Поскольку классы байтовых потоков берут свое начало с классов `InputStream` и `OutputStream`, с них и начнем обсуждение.

Класс `InputStream`

Класс `InputStream` — это абстрактный класс, определяющий модель Java байтового потокового ввода. Он реализует интерфейсы `AutoCloseable` и `Closeable`. При ошибках ввода-вывода большинство методов этого класса передает исключение `IOException`. (Сюда не входят методы `mark()` и `markSupported()`.) В табл. 19.2 перечислены методы класса `InputStream`.

Таблица 19.2. Методы, определенные в классе `InputStream`

Метод	Описание
`int available()`	Возвращает количество байтов ввода, которые доступны в данный момент для чтения
`void close()`	Закрывает источник ввода. Последующие попытки чтения передадут исключение `IOException`
`void mark(int` *`колБайтов`*`)`	Помещает метку в текущую точку входного потока, которая остается корректной до тех пор, пока не будет прочитано *`колБайтов`* байт
`boolean markSupported()`	Возвращает значение `true`, если методы `mark()` и `reset()` поддерживаются вызывающим потоком
`int read()`	Возвращает целочисленное представление следующего доступного байта в потоке. При достижении конца файла возвращается значение `-1`
`int read(byte` *`буфер`*`[])`	Пытается читать до *`колБайтов`* в *`буфер`*, возвращая количество успешно прочитанных байтов. По достижении конца файла возвращает значение `-1`
`int read(byte` *`буфер`*`[], int` *`смещение`*`, int` *`колБайтов`*`)`	Пытается читать до *`колБайтов`* в *`буфер`*, начиная с *`буфер[смещение]`* и возвращая количество успешно прочитанных байтов. По достижении конца файла возвращает значение `-1`
`void reset()`	Сбрасывает входной указатель в ранее установленную метку
`long skip(long` *`колБайтов`*`)`	Игнорирует (т.е. пропускает) *`колБайтов`* байт ввода, возвращая количество действительности проигнорированных байтов

На заметку! Большинство методов, описанных в табл. 19.1, реализуется производными классами класса `InputStream`. Методы `mark()` и `reset()` — исключения; обратите внимание на их использование или отсутствие такового в каждом производном классе и следующих обсуждениях.

Класс `OutputStream`

Класс `OutputStream` — это абстрактный класс, определяющий потоковый байтовый вывод. Реализует интерфейсы `AutoCloseable`, `Closeable` и `Flushable`. Большинство методов этого класса возвращает `void` и передает исключение `IOException` в случае ошибок ввода-вывода. В табл. 19.3 перечислены методы класса `OutputStream`.

На заметку! Большинство методов, описанных в табл. 19.2 и 19.3, реализовано подклассами классов `InputStream` и `OutputStream`. Методы `mark()` и `reset()` являются исключениями; имейте это в виду, когда ниже речь пойдет о каждом подклассе.

Таблица 19.3. Методы, определенные в классе `OutputStream`

Метод	Описание
`int close()`	Закрывает выходной поток. Последующие попытки записи передадут исключение `IOException`
`void flush()`	Финализирует выходное состояние, очищая все буферы, т.е. очищает буферы вывода
`void write(int b)`	Записывает единственный байт в выходной поток. Обратите внимание на то, что параметр имеет тип `int`, а это позволяет вызывать метод `write()` с выражениями без необходимости приведения их обратно к типу `byte`
`void write(byte буфер[])`	Записывает полный массив байтов в выходной поток
`void write(byte буфер[], int колБайтов)`	Записывает диапазон из *колБайтов* байт из массива *буфер* начиная с *буфер[смещение]*

Класс `FileInputStream`

Класс `FileInputStream` создает объект класса `InputStream`, который вы можете использовать для чтения байтов из файла. Так выглядят два его наиболее часто используемых конструктора.

```
FileInputStream(String путьКфайлу)
FileInputStream(File объектФайла)
```

Каждый из них может передать исключение `FileNotFoundException`. Здесь *путьКфайлу* — полное путевое имя файла, а *объектФайла* — объект класса `File`, описывающий файл. В следующем примере создается два объекта класса `FileInputStream`, использующих один и тот же дисковый файл и оба эти конструктора.

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")
File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

Хотя первый конструктор, вероятно, используется чаще, второй позволяет подробно исследовать файл с помощью методов класса `File`, прежде чем присоединять его к входному потоку. При создании объект класса `FileInputStream` открывается также и для чтения. Класс `FileInputStream` переопределяет шесть методов абстрактного класса `InputStream`. Методы `mark()` и `reset()` не переопределяются, и все попытки использовать метод `reset()` с объектом класса `FileInputStream` приводят к передаче исключения `IOException`.

Следующий пример показывает, как прочесть один байт, массив байтов и диапазон из массива байтов. Также он иллюстрирует использование метода `available()` для определения оставшегося количества байтов, а также метода `skip()` — для пропуска нежелательных байтов. Программа читает свой собственный исходный файл, который должен присутствовать в текущем каталоге. Обратите внимание на то, что здесь используется новый оператор JDK 7 *try-с-ресурсами* для автоматического закрытия файла, когда он больше не нужен.

```
// Демонстрация применения FileInputStream.
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.

import java.io.*;

class FileInputStreamDemo {
    public static void main(String args[]) {
        int size;

        // Для закрытия потока используется try-с-ресурсами.
        try ( FileInputStream f =

              new FileInputStream("FileInputStreamDemo.java") ) {

            System.out.println("Total Available Bytes: " +
                               (size = f.available()));

            int n = size/40;
            System.out.println("First " + n +
                               " bytes of the file one read() at a
                               time");
            for (int i=0; i < n; i++) {
                System.out.print((char) f.read());
            }

            System.out.println("\nStill Available: " + f.available());

            System.out.println("Reading the next " + n +
                               " with one read(b[])");
            byte b[] = new byte[n];
            if (f.read(b) != n) {
                System.err.println("couldn't read " + n + " bytes.");
            }

            System.out.println(new String(b, 0, n));
            System.out.println("\nStill Available: " +
                               (size = f.available()));
            System.out.println("Skipping half of remaining bytes
                               with skip()");
            f.skip(size/2);
            System.out.println("Still Available: " + f.available());

            System.out.println("Reading " + n/2 +
                               " into the end of array");
            if (f.read(b, n/2, n/2) != n/2) {
                System.err.println("couldn't read " + n/2 + " bytes.");
            }

            System.out.println(new String(b, 0, b.length));
            System.out.println("\nStill Available: " + f.available());
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}
```

Так выглядит вывод этой программы.

```
Total Available Bytes: 1785
First 44 bytes of the file one read() at a time
// Demonstrate FileInputStream.
// This pr
```

```
Still Available: 1741
Reading the next 44 with one read(b[])
ogram uses try-with-resources. It requires J

Still Available: 1697
Skipping half of remaining bytes with skip()
Still Available: 849
Reading 22 into the end of array
ogram uses try-with-rebyte[n];
      if (

Still Available: 827
```

Этот несколько надуманный пример демонстрирует чтение тремя способами, пропуск ввода и проверку количества доступных данных в потоке.

На заметку! Этот и другие примеры этой главы обрабатывают все исключения ввода-вывода, которые могли бы произойти, как описано в главе 13. (Более подробная информация по этой теме приведена в главе 13.)

Класс `FileOutputStream`

Класс `FileOutputStream` создает объект класса `OutputStream`, который вы можете использовать для записи байтов в файл. Он реализует интерфейсы `AutoCloseable`, `Closeable` и `Flushable`. Вот четыре его наиболее часто используемых конструктора.

```
FileOutputStream(String путькфайлу)
FileOutputStream(File объектФайла)
FileOutputStream(String путькфайлу, boolean добавить)
FileOutputStream(File объектФайла, boolean добавить)
```

Они могут передать исключение `FileNotFoundException`. Здесь *путькфайлу* — полное путевое имя файла, а *объектФайла* — объект класса `File`, описывающий файл. Если параметр *добавить* содержит значение `true`, файл открывается в режиме добавления.

Создание объекта класса `FileOutputStream` не зависит от того, существует ли указанный файл. Класс `FileOutputStream` создает его перед открытием, когда вы создаете объект. В случае попытки открытия файла, доступного только для чтения, будет передано исключение.

В следующем примере создается буфер байтов. Сначала создается объект класса `String`, а затем используется метод `getBytes()` для извлечения его эквивалента в виде байтового массива. Затем создается три файла. Первый — `file1.txt` — будет содержать каждый второй байт примера. Второй — `file2.txt` — полный набор байтов. Третий — `file3.txt` — будет содержать только последнюю четверть.

```
// Демонстрация применения FileOutputStream.
// Эта программа использует традиционный подход закрытия файла.

import java.io.*;

class FileOutputStreamDemo {
    public static void main(String args[]) {
        String source = "Now is the time for all good men\n" +
                        " to come to the aid of their country\n" +
                        " and pay their due taxes.";
        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");

            // запись в первый файл
            for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

            // запись во второй файл
            f1.write(buf);

            // запись в третий файл
            f2.write(buf, buf.length-buf.length/4, buf.length/4);
        } catch(IOException e) {
            System.out.println("An I/O Error Occurred");
        } finally {
            try {
                if(f0 != null) f0.close();
            } catch(IOException e) {
                System.out.println("Error Closing file1.txt");
            }
            try {
                if(f1 != null) f1.close();
            } catch(IOException e) {
                System.out.println("Error Closing file2.txt");
            }
            try {
                if(f2 != null) f2.close();
            } catch(IOException e) {
                System.out.println("Error Closing file3.txt");
            }
        }
    }
}
```

Так будет выглядеть содержимое каждого файла после выполнения этой программы. Сначала будет представлен файл `file1.txt`.

```
Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.
Затем файл file2.txt.
Now is the time for all good men
to come to the aid of their country
and pay their due taxes.
```

И наконец, файл `file3.txt`.

```
nd pay their due taxes.
```

Как видно из комментариев вверху, приведенная выше программа демонстрирует пример использования традиционного подхода закрытия файла, когда он больше не нужен. Этот подход применялся всеми версиями Java до JDK 7 и широко используется в устаревшем коде. Как можно заметить, здесь есть немного довольно неуклюжего кода, требуемого для явного вызова метода `close()`, поскольку каждый его вызов может передать исключение `IOException`, если операция закрытия потерпит неудачу. Эта программа может быть существенно улучшена при использовании нового оператора *try-с-ресурсами*. Вот, для сравнения, его переделанная версия. Обратите внимание: она намного короче и понятнее.

```
// Демонстрация применения FileOutputStream.
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.
```

```
import java.io.*;

class FileOutputStreamDemo {
    public static void main(String args[]) {
        String source = "Now is the time for all good men\n" +
                        " to come to the aid of their country\n" +
                        " and pay their due taxes.";
        byte buf[] = source.getBytes();

        // Использование Try-с-ресурсами для закрытия файлов.
        try (FileOutputStream f0 = new FileOutputStream("file1.txt");
             FileOutputStream f1 = new FileOutputStream("file2.txt");
             FileOutputStream f2 = new FileOutputStream("file3.txt") )
        {

            // запись в первый файл
            for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

            // запись во второй файл
            f1.write(buf);

            // запись в третий файл
            f2.write(buf, buf.length-buf.length/4, buf.length/4);
        } catch(IOException e) {
            System.out.println("An I/O Error Occurred");
        }
    }
}
```

Класс ByteArrayInputStream

Класс `ByteArrayInputStream` — реализация входного потока, использующего байтовый массив в качестве источника данных. Этот класс имеет два конструктора, каждый из которых требует байтового массива в качестве источника данных.

```
ByteArrayInputStream(byte массив[ ])
ByteArrayInputStream(byte массив[ ], int начало, int колБайтов)
```

Здесь *массив* — источник данных. Второй конструктор создает объект класса `InputStream` из подмножества байтового массива, который начинается с символа в позиции, указанной в *начало*, и длиной *колБайтов*.

Метод `close()` не имеет никакого влияния на класс `ByteArrayInputStream`. Поэтому нет необходимости вызывать метод `close()` для объекта класса `ByteArrayInputStream`, но ошибкой это не будет.

В следующем примере создается пара объектов класса `ByteArrayInputStream`, которая инициализируется байтами, представляющими английский алфавит.

```
// Демонстрация применения ByteArrayInputStream.
import java.io.*;

class ByteArrayInputStreamDemo {
    public static void main(String args[]) throws IOException {
        String tmp = "abcdefghijklmnopqrstuvwxyz";
        byte b[] = tmp.getBytes();

        ByteArrayInputStream input1 = new ByteArrayInputStream(b);
        ByteArrayInputStream input2 = new ByteArrayInputStream(b,0,3);
    }
}
```

Объект input1 содержит полный алфавит в нижнем регистре, в то время как объект input2 — только первые три буквы.

Класс ByteArrayInputStream реализует методы mark() и reset(). Однако если метод mark() не вызывается, то метод reset() устанавливает указатель потока в его начало — в начало байтового массива, переданного конструктору. Следующий пример показывает, как использовать метод reset() для чтения одного и того же ввода дважды. В этом случае программа читает и выводит буквы "abc" сначала в нижнем регистре, а затем 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();
        }
    }
}
```

Код в этом примере сначала читает каждый символ потока и печатает его, как он есть, — в нижнем регистре. Затем он сбрасывает поток и начинает чтение заново, на этот раз преобразуя перед выводом каждый символ в верхний регистр. Вывод получается таким.

```
abc
ABC
```

Класс ByteArrayOutputStream

Класс ByteArrayOutputStream — это реализация потока вывода, использующего байтовый массив в качестве места назначения. Класс ByteArrayOutputStream имеет два конструктора, показанных ниже.

```
ByteArrayOutputStream()
ByteArrayOutputStream(int колБайтов)
```

В первой форме создается буфер в 32 байт. Во втором создается буфер указанного в параметре *колБайтов* размера. Буфер хранится в защищенном поле buf класса ByteArrayOutputStream. Размер буфера увеличивается автоматически по мере необходимости. Количество байтов, содержащихся в буфере, хранится в защищенном поле count класса ByteArrayOutputStream.

Метод close() не имеет никакого влияния на класс ByteArrayOutputStream. Поэтому нет необходимости вызывать его для объекта класса ByteArrayOutputStream, но ошибкой это не будет.

В следующем примере демонстрируется использование класса ByteArrayOutputStream.

```
// Демонстрация применения ByteArrayOutputStream.
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.

import java.io.*;

class ByteArrayOutputStreamDemo {
    public static void main(String args[]) {
        ByteArrayOutputStream f = new ByteArrayOutputStream();
        String s = "This should end up in the array";
        byte buf[] = s.getBytes();

        try {
            f.write(buf);
        } catch(IOException e) {
            System.out.println("Error Writing to Buffer");
            return;
        }

        System.out.println("Buffer as a string");
        System.out.println(f.toString());
        System.out.println("Into array");
        byte b[] = f.toByteArray();
        for (int i=0; i<b.length; i++) System.out.print((char) b[i]);

        System.out.println("\nTo an OutputStream()");

        // Использование Try-с-ресурсами для управления
        // файловыми потоками.
        try ( FileOutputStream f2 = new FileOutputStream("test.txt") )
        {
            f.writeTo(f2);
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
            return;
        }

        System.out.println("Doing a reset");
        f.reset();

        for (int i=0; i\<3; i++) f.write('X');

        System.out.println(f.toString());
    }
}
```

Запустив эту программу, вы получите следующий вывод. Обратите внимание: после вызова метода `reset()` выводится вначале три буквы `'X'`.

```
Buffer as a string
This should end up in the array
Into array
This should end up in the array
To an OutputStream()
Doing a reset
XXX
```

В этом примере для записи содержимого файла `test.txt` используется удобный метод `writeTo()`. Просмотр файла `test.txt`, созданного в предыдущем примере, покажет результат, который следовало ожидать.

```
This should end up in the array
```

Фильтруемые потоки байтов

Фильтруемые потоки байтов — это просто оболочки вокруг входных или выходных потоков, обеспечивающие некоторые дополнительные функциональные возможности. Эти потоки обычно доступны методам, которые ожидают обобщенного потока, являющегося суперклассом фильтрованного потока. Типичными расширениями являются буферизация, преобразование символов и базовых данных. Фильтруемые потоки байтов — это `FilterInputStream` и `FilterOutputStream`. Их конструкторы показаны ниже.

```
FilterOutputStream(OutputStream os)
FilterInputStream(InputStream is)
```

Методы, представленные в этих классах, идентичны методам классов `InputStream` и `OutputStream`.

Буферизуемые потоки байтов

Для байтовых потоков буферизованные потоки расширяют класс фильтруемого потока, добавляя к нему буфер в памяти. Этот буфер позволяет Java выполнять операции ввода-вывода более чем по одному байту за раз, тем самым повышая производительность. Благодаря доступности буфера, возможны пропуск, маркировка и сброс потока. Буферизованные байтовые потоки имеют классы `BufferedInputStream` и `BufferedOutputStream`. Класс `PushbackInputStream` также реализует буферизованный поток.

Класс `BufferedInputStream`

Буферизация ввода-вывода — очень распространенный способ оптимизации производительности. Класс `BufferedInputStream` позволяет заключить в оболочку любой поток класса `InputStream` и достичь увеличения производительности. Класс `BufferedInputStream` имеет два конструктора.

```
BufferedInputStream(InputStream входнойПоток)
BufferedInputStream(InputStream входнойПоток, int размерБуфера)
```

Первая форма создает буферизованный поток, использующий размер буфера по умолчанию. Во второй форме размер буфера указывается параметром *размерБуфера*. Рекомендуется использовать размеры буфера, кратные размеру страницы памяти, дисковому блоку и т.п., — это окажет существенное положительное влияние на производительность. Однако, с другой стороны, это зависит от реализации. Необязательный размер буфера обычно зависит от принимающей операционной системы, объема доступной памяти и конфигурации машины. Чтобы добиться эффективного использования буферизации, не обязательно погружаться во все эти сложности. Было бы неплохо установить размер буфера для потока ввода-вывода в 8192 байт или даже меньше. Таким образом, низкоуровневая система сможет читать блоки данных с диска или из сети и сохранять результат в вашем буфере. То есть, даже если вы читаете данные по одному байту из объекта класса `InputStream`, то большую часть времени будете иметь дело с быстрой памятью.

Следующий пример моделирует ситуацию, в которой мы можем использовать метод `mark()` для запоминания места во входном потоке, чтобы позднее вернуться к нему с помощью метода `reset()`. Этот пример разбирает поток, находя в нем конструкцию HTML, указывающую на символ авторских прав. Такая ссылка начинается с амперсанда (`&`), заканчивается точкой с запятой (`;`) и не содержит какие-либо внутренние пробелы. Пример ввода содержит два амперсанда, чтобы показать случай, когда метод `reset()` срабатывает, а когда — нет.

```
// Использование буферизованного ввода.
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.

import java.io.*;

class BufferedInputStreamDemo {
    public static void main(String args[]) {
        String s = "This is a &copy; copyright symbol " +
                   "but this is &copy not.\n";
        byte buf[] = s.getBytes();

        ByteArrayInputStream in = new ByteArrayInputStream(buf);
        int c;
        boolean marked = false;

        // Использование Try-с-ресурсами для управления файлами.
        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("I/O Error: " + e);
        }
    }
}
```

Обратите внимание на то, что этот пример использует метод `mark(32)`, что сохраняет метку для чтения следующих 32 байт (чего достаточно для любых ссылок на сущности). Вот как выглядит вывод, создаваемый этой программой.

```
This is a (c) copyright symbol but this is &copy not.
```

Класс BufferedOutputStream

Класс BufferedOutputStream подобен любому классу OutputStream, за исключением дополнительного метода flush(), используемого для обеспечения записи данных в буферизируемый поток. Поскольку назначение класса BufferedOutputStream — увеличивать производительность за счет сокращения количества физических записей данных, вам может понадобиться вызывать метод flush(), чтобы инициировать немедленную запись всех данных из буфера.

В отличие от буферизованного ввода, буферизованный вывод не предоставляет дополнительной функциональности. Буферы вывода в Java нужны для повышения производительности. Вот два доступных конструктора этого класса.

```
BufferedOutputStream(OutputStream выходнойПоток)
BufferedOutputStream(OutputStream выходнойПоток, int размерБуфера)
```

Первая форма создает буферизованный поток, используя размер буфера по умолчанию. Во второй форме размер буфера передается параметром *размерБуфера*.

Класс PushbackInputStream

Одним из новшеств в буферизации является реализация "вталкивания" (pushback). "Вталкивание" используется с потоками ввода, чтобы обеспечить чтение байта с последующим его возвратом (т.е. "втолкнуть") в поток. Класс PushbackInputStream реализует эту идею. Он представляет механизм для того, чтобы "заглянуть" во входной поток и увидеть, что оттуда поступит в следующий раз, не извлекая информации.

Класс PushbackInputStream имеет следующие конструкторы.

```
PushbackInputStream(InputStream входнойПоток)
PushbackInputStream(InputStream входнойПоток, int колБайтов)
```

Первая форма создает объект потока, позволяющий вернуть один байт во входной поток. Вторая форма создает поток, оснащенный буфером "вталкивания" длиной *колБайтов*. Это позволяет вернуть во входной поток множество байтов.

Помимо знакомых уже методов класса InputStream, класс PushbackInputStream предлагает метод unread().

```
void unread(int б)
void unread(byte буфер[ ])
void unread(byte буфер, int смещение, int количБайтов)
```

Первая форма вталкивает в поток младший байт *б*. После этого он вновь будет следующим байтом, возвращаемым последующим вызовом метода read(). Вторая форма вталкивает байты в *буфер*. Третья же форма вталкивает *количБайтов* байт, начиная с позиции *смещение*, в *буфер*. Исключение IOException передается в случае попытки втолкнуть байт, когда буфер переполнен.

Рассмотрим пример, демонстрирующий, как синтаксический анализатор языка программирования может использовать класс PushbackInputStream и метод unread(), чтобы справиться с различием между операциями сравнения (==) и присваивания (=).

```
// Демонстрация применения unread().
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.

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("I/O Error: " + e);
        }
    }
}
```

Ниже показан вывод этого примера. Обратите внимание на то, что "==" заменяется на ".eq.", а "=" — на "<-".

```
if (a .eq. 4) a<- 0;
```

Внимание! Класс `PushbackInputStream` обладает побочным эффектом — он делает невозможным использование методов `mark()` и `reset()` потока класса `InputStream`, использованного для его создания. Применяйте метод `markSupported()`, чтобы проверить каждый поток на предмет возможности использования методов `mark()` и `reset()`.

Класс `SequenceInputStream`

Класс `SequenceInputStream` позволяет соединять вместе несколько экземпляров класса `InputStream`. Создание объекта класса `SequenceInputStream` отличается от создания объекта класса `InputStream`. Конструктор класса `SequenceInputStream` принимает в качестве аргумента либо пару объектов класса `InputStream`, либо интерфейс `Enumeration` из объекта класса `InputStream`.

```
SequenceInputStream(InputStream первый, InputStream второй)
SequenceInputStream(Enumeration <? extends InputStream> перечПотоков)
```

Во время работы класс выполняет запросы на чтение из первого объекта класса `InputStream` и до конца, а затем переключается на второй. В случае интерфейса `Enumeration` работа продолжится по всем объектам классам `InputStream`, пока не будет достигнут конец последнего. По достижении конца каждого файла, связанный с ним поток закрывается. Закрытие потока, созданного объектом класса `SequenceInputStream`, приводит к закрытию всех открытых потоков.

Вот простой пример использования класса `SequenceInputStream` для вывода содержимого двух файлов. В демонстрационных целях эта программа использует традиционную методику закрытия файла. В качестве упражнения вы могли бы попробовать изменить его так, чтобы использовать оператор *try-с-ресурсами*.

```
// Демонстрация последовательного ввода.
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.
```

```
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("Error Opening File.");
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
        } finally {
            try {
             .  input.close();
            } catch(IOException e) {
                System.out.println("Error Closing SequenceInputStream");
            }
        }
    }
}
```

Этот пример создает объект класса `Vector` и затем добавляет к нему три имени файла. Затем этот вектор с именами передается классу `InputStreamEnumerator`, предназначенному служить оболочкой вектора, в которой элементы возвращаются не в виде имен файлов, а в виде открытых объектов класса `FileInputStream`, созданных по этим именам. Объект класса `SequenceInputStream` открывает каждый файл по очереди, и таким образом этот пример выводит содержимое этих файлов.

Обратите внимание на то, что если файл в методе `nextElement()` не может быть открыт, возвращается значение `null`. В результате передается исключение `NullPointerException`, обрабатываемое в функции `main()`.

Класс PrintStream

Класс PrintStream предоставляет все возможности вывода, которыми мы пользуемся с дескриптором System.out файла класса System с самого начала нашей книги. Это делает класс PrintStream одним из наиболее часто используемых классов Java. Он реализует интерфейсы Appendable, AutoCloseable, Closeable и Flushable.

Класс PrintStream определяет несколько конструкторов. Для начала рассмотрим перечисленные ниже.

```
PrintStream(OutputStream выходнойПоток)
PrintStream(OutputStream выходнойПоток, boolean сбросПриНовойСтроке)
PrintStream(OutputStream выходнойПоток, boolean сбросПриНовойСтроке,
            String наборСимволов) throws UnsupportedEncodingException
```

Здесь *выходнойПоток* указывает открытый объект класса OutputStream, который будет принимать вывод. Параметр *сбросПриНовойСтроке* управляет тем, будет ли выходной буфер автоматически сбрасываться при каждой записи символа новой строки (\n), записи байтового массива либо вызове метода println(). Если аргумент *сбросПриНовойСтроке* равен значению true, происходит автоматический сброс. Если же он равен значению false, сброс будет неавтоматическим. Первый конструктор не включает автоматический сброс. Вы можете задать кодировку символов, передав ее имя в параметре *наборСимволов*.

Следующий набор конструкторов предоставляет простые способы создания объекта класса PrintStream, который пишет свой вывод в файл.

```
PrintStream(File выходнойФайл) throws FileNotFoundException
PrintStream(File выходнойФайл, String наборСимволов)
    throws FileNotFoundException, UnsupportedEncodingException
PrintStream(String имяВыходногоФайла) throws FileNotFoundException
PrintStream(String имяВыходногоФайла, String наборСимволов)
    throws FileNotFoundException, UnsupportedEncodingException
```

Они позволяют создавать объект класса PrintStream на основе объекта класса File либо имени файла. В любом случае файл создается автоматически. Любой существующий файл с тем же именем уничтожается. Будучи созданным, объект класса PrintStream управляет всем выводом в указанный файл. Кодировку символов можно указать в параметре *наборСимволов*.

Класс PrintStream поддерживает методы print() и println() для всех типов, включая тип Object. Если аргумент не относится к элементарному типу, то методы класса PrintStream вызывают метод toString() объекта, а затем отображают его результат.

Не так давно (с появлением версии JDK 5) в класс PrintStream был добавлен метод printf(). Он позволяет задать точный формат вывода записываемых данных. Метод printf() использует класс Formatter (описанный в главе 18) для форматирования данных. Затем он выводит эти данные в вызывающий поток. Хотя форматирование может выполняться вручную за счет непосредственного использования класса Formatter, все же метод printf() существенно упрощает процесс. Он является аналогом функции C/C++ printf(), облегчая преобразование существующего кода C/C++ в Java. Откровенно говоря, метод printf() — весьма полезное дополнение к Java API, поскольку значительно упрощает вывод форматированных данных на консоль. Метод printf() имеет следующие общие формы.

```
PrintStream printf(String формСтрока, Object ... аргументы)
PrintStream printf(Locale регион, String формСтрока, Object ...
аргументы)
```

Первая версия записывает *аргументы* в стандартный вывод в формате, указанном *формСтрока*, используя локальные установки по умолчанию. Вторая версия

позволяет указать региональные данные. Обе возвращают вызывающий объект класса PrintStream.

В общем случае метод printf() подобен методу format(), который определен в классе Formatter. Параметр *формСтрока* состоит из элементов двух типов. Первый тип содержит символы, которые просто копируются в выходной буфер, второй тип – спецификаторы формата, определяющие способ отображения последующих аргументов – *аргументы*. За полной информацией о форматированном выводе, включая описание спецификаторов формата, обращайтесь к описанию класса Formatter в главе 18.

Поскольку поток System.out имеет тип PrintStream, вы можете вызывать метод printf() с потоком System.out. Поэтому метод printf() может служить в качестве замены метода println(), когда необходимо выдавать на консоль форматированный вывод. Например, в следующей программе метод printf() используется для вывода числовых значений в различных форматах. До JDK 5 такое форматирование требовало существенной работы. С появлением метода printf() оно значительно упростилось.

```
// Демонстрация применения printf().

class PrintfDemo {
    public static void main(String args[]) {
        System.out.println("Ниже следуют некоторые числовые значения " +
                           "в различных форматах.\n");

        System.out.printf("Различные целочисленные форматы: ");
        System.out.printf("%d %(d %+d %05d\n", 3, -3, 3, 3);

        System.out.println();
        System.out.printf("Формат с плавающей точкой по умолчанию:
                          %f\n", 1234567.123);
        System.out.printf("Плавающая точка с запятыми: %,f\n",
                           1234567.123);
        System.out.printf("Отрицательная плавающая точка по умолчанию:
                           %,f\n", -1234567.123);
        System.out.printf("Параметры отрицательной плавающей точки:
                           %,(f\n", -1234567.123);

        System.out.println();

        System.out.printf("Строка из положительных и отрицательных
                           значений:\n");
        System.out.printf("% ,.2f\n% ,.2f\n",
    }
}
```

Вывод этой программы.

```
Ниже следуют некоторые числовые значения в различных форматах.

Различные целочисленные форматы: 3 (3) +3 00003

Формат с плавающей точкой по умолчанию: 1234567.123000
Плавающая точка с запятыми: 1,234,567.123000
Отрицательная плавающая точка по умолчанию: -1,234,567.123000
Параметры отрицательной плавающей точки: (1,234,567.123000)

Строка из положительных и отрицательных значений:
1,234,567.12
-1,234,567.12
```

В классе PrintStream определен также метод format(). Вот его общие формы.

```
PrintStream format(String формСтрока, Object ... аргументы)
PrintStream format(Locale регион, String формСтрока,
                                    Object ... аргументы)
```

Он работает точно так же, как метод `printf()`.

Классы DataOutputStream и DataInputStream

Эти классы позволяют писать или читать элементарные данные в поток и из него. Они реализуют интерфейсы `DataOutput` и `DataInput` соответственно. Эти интерфейсы определяют методы, преобразующие элементарные значения в форму последовательности байтов. Такие потоки облегчают сохранение в файле двоичных данных, таких как целочисленные значения или значения с плавающей точкой. Рассмотрим здесь и то, и другое.

Класс `DataOutputStream` расширяет класс `FilterOutputStream`, который, в свою очередь, расширяет класс `OutputStream`. Кроме реализации интерфейса `DataOutput`, класс `DataOutputStream` реализует также интерфейсы `AutoCloseable`, `Closeable` и `Flushable`. В классе `DataOutputStream` определен следующий конструктор.

```
DataOutputStream(OutputStream выходнойПоток)
```

Здесь *выходнойПоток* определяет выходной поток, в который будут записаны данные. Когда поток класса `DataOutputStream` закрывается (при вызове метода `close()`), основной поток, определенный аргументом *выходнойПоток*, также закрывается автоматически.

Класс `DataOutputStream` поддерживает все методы, определенные его суперклассами. Однако он реализует методы, определенные интерфейсом `DataOutput`, которые и делают его интересным. Интерфейс `DataOutput` определяет методы, преобразующие значения элементарных типов в последовательности байтов, а затем записывающие их в лежащий в основе поток. Вот образцы этих методов.

```
final void writeDouble(double значение) throws IOException
final void writeBoolean(boolean значение) throws IOException
final void writeInt(int значение) throws IOException
```

Здесь *значение* — это значение, записываемое в поток.

Класс `DataInputStream` — это дополнение класса `DataOutputStream`. Он расширяет класс `FilterInputStream`, который, в свою очередь, расширяет класс `InputStream`. Кроме реализации интерфейса `DataInput`, класс `DataInputStream` реализует также интерфейсы `AutoCloseable` и `Closeable`. Он определяет только один следующий конструктор.

```
DataInputStream(InputStream входнойПоток)
```

Здесь *входнойПоток* определяет входной поток, откуда будут читаться данные. Когда поток класса `DataInputStream` закрывается (при вызове метода `close()`), основной поток, определенный аргументом *входнойПоток*, также закрывается автоматически.

Как и класс `DataOutputStream`, класс `DataInputStream` поддерживает все методы своих суперклассов, наряду с методами, определенными интерфейсом `DataInput`, что и делает его уникальным. Эти методы читают последовательность байтов и преобразуют их в значения элементарных типов. Ниже показаны образцы этих методов.

```
final double readDouble( ) throws IOException
final boolean readBoolean( ) throws IOException
final int readInt( ) throws IOException
```

В следующей программе демонстрируется использование классов `DataOutputStream` и `DataInputStream`.

```
// Демонстрация применения DataInputStream и DataOutputStream.
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.

import java.io.*;

class DataIODemo {
    public static void main(String args[])

    // Сначала запись данных.
    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("Cannot Open Output File");
            return;
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
        }

        // Теперь прочитать данные назад.
        try ( DataInputStream din =
              new DataInputStream(new FileInputStream("Test.dat")) )
        {
            double d = din.readDouble();
            int i = din.readInt();
            boolean b = din.readBoolean();

            System.out.println("Вот значения: " +
                               d + " " + i + " " + b);

        } catch(FileNotFoundException e) {
            System.out.println("Cannot Open Input File");
            return;
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}
```

Вывод приведен ниже.

```
Вот значения: 98.6 1000 true
```

Класс RandomAccessFile

Класс `RandomAccessFile` инкапсулирует файл произвольного доступа. Он не наследуется от класса `InputStream` или `OutputStream`. Вместо этого он реализует интерфейсы `DataInput` и `DataOutput`, которые определяют базовые методы ввода-вывода. Он также реализует интерфейсы `AutoCloseable` и `Closeable`. Класс `RandomAccessFile` отличает его поддержка запросов на позиционирование, т.е. вы можете установить указатель файла в любое место в пределах этого файла. Этот класс включает следующие два конструктора.

```
RandomAccessFile(File объектФайла, String доступ)
    throws FileNotFoundException
RandomAccessFile(String имяфайла, String доступ)
    throws FileNotFoundException
```

В первой форме *объектФайла* задает открываемый файл как объект класса `File`. Во второй форме имя файла передается параметром *имяфайла*. В обоих

случаях *доступ* определяет тип доступа. Если он равен "r", то файл может быть прочитан, но не может быть записан. Если "rw", то файл открывается в режиме чтения-записи. Если же *доступ* равен "rws", то файл открывается для операций чтения-записи и каждое изменение данных файла или его метаданных немедленно записывается на физическое устройство. Метод seek(), показанный ниже, используется для установки текущей позиции указателя внутри файла.

```
void seek(long новПоз) throws IOException
```

Здесь *новПоз* указывает новую позицию в байтах файлового указателя от начала файла. После вызова метода seek() следующая операция чтения или записи выполняется в этой новой позиции.

Класс RandomAccessFile реализует стандартные методы ввода и вывода, которые вы можете использовать для чтения и записи файлов произвольного доступа. Кроме того, он включает несколько дополнительных методов. Одним из них является метод setLength(). Его сигнатура такова.

```
void setLength(long длина) throws IOException
```

Этот метод устанавливает длину вызывающего файла равной указанному значению *длина*. Метод может использоваться для удлинения или укорачивания файла. Если файл удлиняется, его добавочная порция является неопределенной.

Символьные потоки

В то время как классы байтовых потоков предоставляют необходимые функциональные возможности для выполнения операций ввода-вывода любого типа, они не могут работать напрямую с символами Unicode. Поскольку одной из главных целей Java является поддержка философии "написано однажды, выполняется везде", понадобилось включить поддержку прямого ввода-вывода для символов. В этом разделе обсудим несколько классов символьного ввода-вывода. Как уже объяснялось, в вершине иерархии символьных потоков находятся абстрактные классы Reader и Writer. С них и начнем.

На заметку! Как было сказано в главе 13, классы символьного ввода-вывода были добавлены в версии Java 1.1. По этой причине вы все еще можете встретить унаследованный код, использующий байтовые потоки там, где более целесообразно было бы применить символьные потоки. Работая с таким кодом, будет неплохо обновить его.

Класс Reader

Класс Reader – абстрактный класс, определяющий символьный потоковый ввод Java. Он реализует интерфейсы AutoCloseable, Closeable и Readable. Все методы этого класса (за исключением метода markSupported()) в случае ошибочных ситуаций передают исключение IOException. В табл. 19.4 представлен краткий обзор методов класса Reader.

Таблица 19.4. Методы, определенные в классе Reader

Метод	Описание
abstract void close()	Закрывает входной поток. Последующие попытки чтения передадут исключение IOException
void mark(int *количСимволов*)	Помещает метку в текущую позицию во входном потоке, которая остается корректной до тех пор, пока не будет прочитано *количСимволов* символов

Окончание табл. 19.4

Метод	Описание
`boolean markSupported()`	Возвращает значение `true`, если поток поддерживает методы `mark()` и `reset()`
`int read()`	Возвращает целочисленное представление следующего доступного символа вызывающего входного потока. При достижении конца файла возвращает значение `-1`
`int read(char буфер[])`	Пытается прочитать до *буфер*`.length` символов в *буфер* и возвращает количество успешно прочитанных символов. При достижении конца файла возвращает значение `-1`
`int read(CharBuffer буфер)`	Пытается читать символы в *буфер* и возвращает фактическое количество успешно прочитанных символов. При достижении конца файла возвращает значение `-1`
`abstract int read(char буфер[], int смещение, int колич-Символов)`	Пытается прочитать до *колич-Символов* символов в *буфер* начиная с *буфер*`[`*смещение*`]`, возвращает количество успешно прочитанных символов. При достижении конца файла возвращает значение `-1`
`boolean ready()`	Возвращает значение `true`, если следующий запрос не будет ожидать. В противном случае возвращает значение `false`
`void reset()`	Сбрасывает указатель ввода в ранее установленную позицию метки
`long skip(long колич-Символов)`	Пропускает *колич-Символов* символов ввода, возвращая количество действительно пропущенных символов

Класс `Writer`

Класс `Writer` — абстрактный класс, определяющий символьный потоковый вывод. Реализует интерфейсы `AutoCloseable`, `Closeable`, `Flushable` и `Appendable`. В случае ошибок все методы этого класса передают исключение `IOException`. Краткий обзор методов класса `Writer` представлен в табл. 19.5.

Таблица 19.5. Методы, определенные в классе `Writer`

Метод	Описание
`Writer append(char символ)`	Добавляет *символ* в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
`Writer append(CharSequence символы)`	Добавляет *символы* в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
`Writer append(CharSequence символы, int начало, int конец)`	Добавляет диапазон *символы*, заданный при помощи *начало* и *конец*`-1`, в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
`abstract void close()`	Закрывает вызывающий поток. Последующие попытки записи передадут исключение `IOException`

Окончание табл. 19.5

Метод	Описание
`abstract void flush()`	Финализирует выходное состояние так, что все буферы очищаются. Иными словами, сбрасывает выходные буферы
`void write(int` *символ*`)`	Записывает единственный символ в вызывающий выходной поток. Обратите внимание на то, что параметр имеет тип `int`, что позволяет вызывать метод `write()` с выражением без необходимости приведения обратно к типу `char`. Однако записываются только младшие 16 бит
`void write(char` *буфер*`[])`	Записывает полный массив символов в вызывающий выходной поток
`abstract void write(char` *буфер*`[], int` *смещение*`, int` *количСимволов*`)`	Записывает диапазон *количСимволов* символов из массива *буфер*, начиная с *буфер*`[`*смещение*`]`, в вызывающий выходной поток
`void write(String` *строка*`)`	Записывает *строка* в вызывающий выходной поток
`void write(String` *строка*`, int` *смещение*`, int` *количСимволов*`)`	Записывает диапазон *количСимволов* символов из строки *строка* начиная с указанного смещения *смещение*

Класс `FileReader`

Это класс, производный от класса `Reader`, который вы можете использовать для чтения содержимого файла. Два наиболее часто используемых его конструктора выглядят так.

```
FileReader(String путькфайлу)
FileReader(File объектФайла)
```

Оба могут передать исключение `FileNotFoundException`. Здесь *путькфайлу* — полное путевое имя файла, а *объектФайла* — объект класса `File`, описывающий файл.

Следующий пример показывает, как можно читать строки из файла и печатать их в стандартный выходной поток. Он читает собственный исходный файл, который должен находиться в текущем каталоге.

```
// Демонстрация применения FileReader.
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.

import java.io.*;

class FileReaderDemo {
    public static void main(String args[]) {
        try ( FileReader fr = new FileReader("FileReaderDemo.java") )
        {
            int c;

            // Читает и отображает файл.
            while((c = fr.read()) != -1) System.out.print((char) c);

        } catch(IOException e) {
```

```
            System.out.println("I/O Error: " + e);
        }
    }
}
```

Класс FileWriter

Этот класс создает объект класса, производного от класса `Writer`, который вы можете применять для записи файла. Рассмотрим его наиболее часто используемые конструкторы.

```
FileWriter(String путькфайлу)
FileWriter(String путькфайлу, boolean добавить)
FileWriter(File объектФайла)
FileWriter(File объектФайла, boolean добавить)
```

Все они могут передавать исключение `IOException`. Здесь *путькфайлу* — полное путевое имя файла, а *объектФайла* — объект класса `File`, описывающий файл. Если *добавить* равно `true`, то вывод добавляется в конец файла.

Создание объекта класса `FileWriter` не зависит от того, существует ли файл. Класс `FileWriter` создаст файл перед его открытием для вывода, когда вы создаете объект. В случае попытки открытия файла, доступного только для чтения, передается исключение `IOException`.

Следующий пример представляет собой версию символьного потока из примера, представленного ранее, когда речь шла о классе `FileOutputStream`. Эта версия создает простой буфер символов, сначала создавая объект класса `String`, а затем используя метод `getChars()` для извлечения эквивалентного символьного массива. Затем она создает три файла. Первый файл, `file.txt`, будет содержать каждый второй символ примера. Второй файл, `file2.txt`, будет хранить полный набор символов. И наконец, третий файл, `file3.txt`, будет содержать только последнюю четверть символов.

```
// Демонстрация применения FileWriter.
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.

import java.io.*;

class FileWriterDemo {
    public static void main(String args[]) throws IOException {
        String source = "Now is the time for all good men\n" +
                        " to come to the aid of their country\n" +
                        " and pay their due taxes.";
        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") )
        {
            // запись в первый файл
            for (int i=0; i < buffer.length; i += 2) {
                f0.write(buffer[i]);
            }

            // запись во второй файл
            f1.write(buffer);

            // запись в третий файл
          f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
```

```
        } catch(IOException e) {
            System.out.println("An I/O Error Occurred");
        }
    }
}
```

Класс CharArrayReader

Класс CharArrayReader — реализация входного потока, использующего символьный массив в качестве источника. Этот класс имеет два конструктора, каждый из которых принимает символьный массив в качестве источника данных.

```
CharArrayReader(char массив[])
CharArrayReader(char массив[], int начало, int количСимволов)
```

Здесь *массив* — входной источник. Второй конструктор создает объект класса, производного от класса Reader, из подмножества символьного массива, начинающегося с символа в позиции, указанной параметром *начало*, и длиной *количСимволов*.

Метод close(), реализованный классом CharArrayReader, не передает исключений. Это связано с тем, что он не может потерпеть неудачу.

В следующем примере используется пара объектов класса CharArrayReaders.

```
// Демонстрация применения CharArrayReader.
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.

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("input1:");
            while((i = input1.read()) != -1) {
                System.out.print((char)i);
            }
            System.out.println();
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
        }

        try (CharArrayReader input2 = new CharArrayReader(c, 0, 5))
        {
            System.out.println("input2:");
            while((i = input2.read()) != -1) {
                System.out.print((char)i);
            }
            System.out.println();
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}
```

Объект input1 создается из полного алфавита в нижнем регистре, в то время как объект input2 содержит только первые пять букв. Вот вывод этой программы.

```
input1:
abcdefghijklmnopqrstuvwxyz
input2:
abcde
```

Класс CharArrayWriter

Класс CharArrayWriter – реализация выходного потока, использующего в качестве места назначения вывода массив. Класс CharArrayWriter имеет два следующих конструктора.

```
CharArrayWriter()
CharArrayWriter(int количСимволов)
```

В первой форме создается буфер с размером по умолчанию. Второй буфер создается с размером, заданным параметром *количСимволов*. Буфер находится в поле buf класса CharArrayWriter. Размер буфера будет при необходимости последовательно увеличиваться. Количество байтов, содержащихся в буфере, находится в поле count того же класса. Оба поля – buf и count – являются защищенными.

Метод close() не имеет никакого влияния на класс CharArrayWriter.

В следующем примере демонстрируется использование класса CharArrayWriter в переделанной программе, рассмотренной ранее, когда речь шла о классе ByteArrayOutputStream. Она приводит к тому же выводу, что и предыдущая версия.

```
// Демонстрация применения CharArrayWriter.
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.

import java.io.*;
class CharArrayWriterDemo {
    public static void main(String args[]) throws IOException {
        CharArrayWriter f = new CharArrayWriter();
        String s = "This should end up in the array";
        char buf[] = new char[s.length()];

        s.getChars(0, s.length(), buf, 0);

        try {
            f.write(buf);
        } catch(IOException e) {
            System.out.println("Error Writing to Buffer");
            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("\nB FileWriter()");

        // Использование Try-с-ресурсами для управления
        // файловыми потоками.
        try ( FileWriter f2 = new FileWriter("test.txt") )
```

```
        {
            f.writeTo(f2);
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
        }

        System.out.println("Выполнение reset()");
        f.reset();

        for (int i=0; i<3; i++) f.write('X');

        System.out.println(f.toString());
    }
}
```

Класс BufferedReader

Класс BufferedReader увеличивает производительность за счет буферизации ввода. У него имеются два конструктора.

```
BufferedReader(Reader входнойПоток)
BufferedReader(Reader входнойПоток, int размерБуфера)
```

Первая форма создает буферизованный символьный поток, используя размер буфера по умолчанию. Во второй форме размер буфера задает *размерБуфера*.

Закрытие объекта класса BufferedReader приводит к закрытию также внутреннего потока, определенного аргументом *входнойПоток*.

Как и в случае с байтовым потоком, буферизованный символьный входной поток также обеспечивает фундамент для поддержки перемещения обратно по потоку в пределах доступного буфера. Для обеспечения этого класс BufferedReader реализует методы mark() и reset(), а метод BufferedReader.markSupported() возвращает значение true.

Следующий пример представляет собой переработанную версию примера с классом BufferedInputStream, показанную выше, но использует символьный поток класса BufferedReader вместо буферизованного байтового потока. Как и ранее, он использует методы mark() и reset() для разбора потока на предмет поиска конструкции HTML с символом авторских прав. Такая ссылка начинается с амперсанда (&) и заканчивается точкой с запятой (;) без внутренних пробелов. Пример ввода содержит два амперсанда, чтобы продемонстрировать случай, когда вызов метода reset() происходит, а когда — нет. Вывод будет таким же, что и раньше.

```
// Использование буферизованного ввода.
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.

import java.io.*;

class BufferedReaderDemo {
    public static void main(String args[]) throws IOException {
        String s = "This is a &copy; copyright symbol " +
                   "but this is &copy not.\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("I/O Error: " + e);
        }
    }
}
```

Класс `BufferedWriter`

Класс `BufferedWriter` — это класс, производный от класса `Writer`, который буферизует вывод. Благодаря применению класса `BufferedWriter`, можно повысить производительность за счет снижения количества операций физической записи в выходное устройство.

Класс `BufferedWriter` имеет следующие два конструктора.

```
BufferedWriter(Writer выходнойПоток)
BufferedWriter(Writer выходнойПоток, int размерБуфера)
```

Первая форма создает буферизованный поток, использующий буфер с размером по умолчанию. Во второй размер буфера передается в параметре *размерБуфера*.

Класс `PushbackReader`

Класс `PushbackReader` позволяет возвращать во входной поток один или более байтов. Это позволяет "заглянуть" во входной буфер. Вот два его конструктора.

```
PushbackReader(Reader входнойПоток)
PushbackReader(Reader входнойПоток, int размерБуфера)
```

Первая форма создает буферизованный поток, позволяющий "втолкнуть" один символ. Во второй форме размер буфера вталкивания передается в параметре *размерБуфера*.

При закрытии объекта класса PushbackReader закрывается также внутренний поток, определенный аргументом *входнойПоток*.

Класс PushbackReader предоставляет метод unread(), который возвращает один или более символов в вызывающий входной поток. Доступны три формы этого метода.

```
void unread(int символ) throws IOException
void unread(char буфер[]) throws IOException
void unread(char буфер[], int смещение, int количСимволов) throws
IOException
```

Первая форма вталкивает символ, переданный в параметре *символ*. Этот символ будет первым, который затем вернет последующий вызов метода read(). Вторая форма возвращает в поток символы из *буфер*. Третья форма вталкивает *количСимволов* символов, начиная со смещения *смещение*, в *буфер*. При попытке возврата символа в полный буфер передается исключение IOException.

Следующая программа представляет собой переделанный пример с классом PushbackInputStream, в котором класс PushbackInputStream заменен классом PushbackReader. Как и ранее, он показывает, как синтаксический анализатор языка программирования может использовать "вталкивание" в поток для обнаружения отличия между операциями сравнения (==) и присваивания (=).

```
// Демонстрация применения unread().
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.

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("I/O Error: " + e);
            }
        }
    }
}
```

Класс PrintWriter

Класс PrintWriter – по сути, символьная версия класса PrintStream. Он реализует интерфейсы Appendable, Closeable и Flushable. Класс PrintWriter имеет несколько конструкторов. Для начала рассмотрим следующие конструкторы.

```
PrintWriter(OutputStream выходнойПоток)
PrintWriter(OutputStream выходнойПоток, boolean сбросПриНовойСтроке)
PrintWriter(Writer выходнойПоток)
PrintWriter(Writer выходнойПоток, boolean сбросПриНовойСтроке)
```

Здесь *выходнойПоток* определяет открытый объект класса OutputStream, который примет вывод. Параметр сбросПриНовойСтроке управляет автоматическим выталкиванием буфера при каждом вызове методов println(), printf() или format(). Если параметр сбросПриНовойСтроке содержит значение true, то происходит автоматическое выталкивание буфера. Если же этот параметр содержит значение false, то автоматического выталкивания не происходит. Конструкторы, которые не принимают параметра сбросПриНовойСтроке, автоматического выталкивания не подразумевают.

Следующий набор конструкторов предоставляет простую возможность создания объекта класса PrintWriter, который пишет свой вывод в файл.

```
PrintWriter(File выходнойФайл) throws FileNotFoundException
PrintWriter(File выходнойФайл, String наборСимволов)
    throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(String имяВыходногоФайла) throws FileNotFoundException
PrintWriter(String имяВыходногоФайла, String наборСимволов)
    throws FileNotFoundException, UnsupportedEncodingException
```

Они позволяют создать объект класса PrintWriter на основе объекта класса File либо имени файла. В любом случае файл создается автоматически. Любой ранее существовавший файл с тем же именем уничтожается. Будучи созданным, объект класса PrintWriter направляет весь вывод в указанный файл. Вы можете задать кодировку символов, передав ее имя в параметре *наборСимволов*.

Класс PrintWriter предоставляет методы print() и println() для всех типов, включая тип Object. Если аргумент не относится к элементарному типу, то методы класса PrintWriter вызывают метод toString() такого объекта, а затем выводят его результат.

Класс PrintWriter также поддерживает метод printf(). Он работает точно так же, как и в классе PrintStream, описанном ранее, – позволяя задать точный формат данных. Метод printf() для класса PrintWriter объявлен следующим образом.

```
PrintWriter printf(String формСтрока, Object ... аргументы)
PrintWriter printf(Locale регион, String формСтрока, Object ... аргументы)
```

Первая версия пишет *аргументы* в стандартный вывод в формате, указанном в *формСтрока*, используя региональные данные по умолчанию. Вторая версия позволяет задать региональные данные. Обе возвращают вызывающий объект класса PrintWriter.

Метод format() также поддерживается. Его общие формы таковы.

```
PrintWriter format(String формСтрока, Object ... аргументы)
PrintWriter format(Locale регион, String формСтрока, Object ... аргументы)
```

Этот метод работает подобно методу printf().

Класс Console

Ранее (в Java SE 6) был добавлен класс Console. Он используется для чтения и записи информации на консоли, если таковая существует, и реализует интерфейс

Flushable. Класс Console, прежде всего, введен для удобства, поскольку большая часть его функциональных возможностей доступна через объекты System.in и System.out. Однако его применение позволяет упростить некоторые виды консольных итераций, особенно при чтении строк с консоли.

Класс Console не поддерживает конструкторов. Его объект получают вызовом метода System.console().

```
static System.console()
```

Если консоль доступна, возвращается ссылка на нее. В противном случае возвращается значение null. Консоль не будет доступна во всех классах, поэтому если возвращается значение null, консольные операции ввода-вывода невозможны.

Класс Console определяет методы, перечисленные в табл. 19.6. Обратите внимание на то, что методы ввода, такие как метод readLine(), передают исключение IOException, когда возникают ошибки ввода. Класс исключения IOError происходит от класса Error и означает сбой ввода-вывода, который происходит вне контроля вашей программы. То есть обычно вы не будете перехватывать исключение IOError. Откровенно говоря, если исключение IOError возникнет в процессе обращения к консоли, обычно это свидетельствует о катастрофическом сбое системы.

Также обратите внимание на методы readPassword(), которые позволяют приложению считывать пароль, не отображая его на экране. Читая пароли, следует "обнулять" как массив, содержащий строку, введенную пользователем, так и массив, содержащий правильный пароль, с которым нужно сравнить первую строку. Это уменьшает шансы вредоносной программы получить пароль при помощи сканирования памяти.

Таблица 19.6. Методы, определенные в классе Console

Метод	Описание
`void flush()`	Выполняет физическую запись буферизованного вывода на консоль
`Console format(String` *формСтрока*`, Object...` *аргументы*`)`	Выводит на консоль *аргументы*, используя формат, указанный в *формСтрока*
`Console printf(String` *формСтрока*`, Object...` *аргументы*`)`	Выводит на консоль *аргументы*, используя формат, указанный в *формСтрока*
`Reader reader()`	Возвращает ссылку на объект класса, производного от класса Reader, соединенный с консолью
`String readLine()`	Читает и возвращает строку, введенную с клавиатуры. Ввод прекращается нажатием клавиши <Enter>. Если достигнут конец входного потока консоли, возвращается значение null. В случае сбоя передается исключение IOError
`String readLine(String` *формСтрока*`, Object...` *аргументы*`)`	Отображает строку приглашения, форматированную в соответствии с *формСтрока* и *аргументы*, затем читает и возвращает строку, введенную с клавиатуры. Ввод прекращается, когда пользователь нажимает клавишу <Enter>. Если достигнут конец входного потока консоли, возвращается значение null. В случае сбоя передается исключение IOError
`char[ ] readPassword()`	Читает и возвращает строку, введенную с клавиатуры. Ввод прекращается нажатием клавиши <Enter>. При этом строка не отображается. Если достигнут конец входного потока консоли, возвращается значение null. В случае сбоя передается исключение IOError

Окончание табл. 19.6

Метод	Описание
`char[ ] readPassword(String` *`формСтрока`*`, Object...` *`аргументы`*`)`	Отображает строку приглашения, форматированную в соответствии с *формСтрока* и *аргументы*, а затем читает и возвращает строку, введенную с клавиатуры. Ввод прекращается, когда пользователь нажимает клавишу <Enter>. При этом строка не отображается. Если достигнут конец входного потока консоли, возвращается значение `null`. В случае сбоя передается исключение `IOError`
`PrintWriter writer()`	Возвращает ссылку на объект класса, производного от класса `Writer`, ассоциированный с консолью

Рассмотрим пример, демонстрирующий класс `Console` в действии.

```
// Демонстрация применения Console.
import java.io.*;

class ConsoleDemo {
    public static void main(String args[]) {
        String str;
        Console con;

        // Получить ссылку на консоль.
        con = System.console();

        // Если нет доступной консоли, выход.
        if(con == null) return;

        // Прочесть строку и отобразить ее.
        str = con.readLine("Введите строку: ");
        con.printf("Вот ваша строка: %s\n", str);
    }
}
```

Вывод этого примера.

```
Введите строку: Это тест.
Вот ваша строка: Это тест.
```

Сериализация

Сериализация — это процесс записи состояния объекта в байтовый поток. Она удобна, когда нужно сохранить состояние вашей программы в области постоянного хранения, такой как файл. Позднее вы можете восстановить эти объекты, используя процесс десериализации.

Сериализация также необходима в реализации *дистанционного вызова методов* (Remote Method Invocation — RMI). RMI позволяет объекту Java на одной машине обращаться к методу объекта Java на другой машине. Объект может быть применен как аргумент этого дистанционного метода. Посылающая машина сериализует объект и передает его. Принимающая машина десериализует его. (Подробнее о RMI будет рассказано в главе 28.)

Предположим, что объект, подлежащий сериализации, ссылается на другие объекты, которые, в свою очередь, имеют ссылки на еще какие-то объекты. Такой набор объектов и отношений между ними формирует ориентированный граф. В этом графе могут присутствовать и циклические ссылки. Иными словами, объект X может содержать ссылку на объект Y, а объект Y — обратную ссылку на X. Объекты также

могут содержать ссылки на самих себя. Средства сериализации и десериализации объектов устроены так, что могут корректно работать во всех этих сценариях. Если вы попытаетесь сериализовать объект, находящийся на вершине такого графа объектов, то все прочие объекты, на которые имеются ссылки, также будут рекурсивно найдены и сериализованы. Аналогично во время процесса десериализации все эти объекты и их ссылки корректно восстанавливаются.

Ниже приведен обзор интерфейсов и классов, поддерживающих сериализацию.

Интерфейс `Serializable`

Только объект, реализующий интерфейс `Serializable`, может быть сохранен и восстановлен средствами сериализации. Интерфейс `Serializable` не содержит никаких членов. Он просто используется для того, чтобы указать, что класс может быть сериализован. Если класс является сериализуемым, все его подклассы также сериализуемы.

Переменные, объявленные как `transient`, не сохраняются средствами сериализации. Не сохраняются также статические переменные.

Интерфейс `Externalizable`

Средства Java для сериализации и десериализации спроектированы так, что бо́льшая часть работы по сохранению и восстановлению состояния объекта выполняется автоматически. Однако бывают случаи, когда программисту нужно управлять этим процессом. Например, может оказаться желательным использовать технологии сжатия и шифрования. Интерфейс `Externalizable` предназначен именно для таких ситуаций.

Интерфейс `Externalizable` определяет следующие два метода.

```
void readExternal(ObjectInput входнойПоток)
    throws IOException, ClassNotFoundException
void writeExternal(ObjectOutput выходнойПоток)
    throws IOException
```

В этих методах *входнойПоток* — это байтовый поток, из которого объект может быть прочитан, а *выходнойПоток* — байтовый поток, куда он записывается.

Интерфейс `ObjectOutput`

Интерфейс `ObjectOutput` расширяет интерфейсы `AutoCloseable` и `DataOutpit`, поддерживает сериализацию объектов. Он определяет методы, показанные в табл. 19.7. Особо отметим метод `writeObject()`. Он вызывается для сериализации объекта. В случае ошибок все методы этого интерфейса передают исключение `IOException`.

Таблица 19.7. Методы, определенные в интерфейсе `ObjectOutput`

Метод	Описание
`void close()`	Закрывает вызывающий поток. Последующие попытки записи передадут исключение `IOException`
`void flush()`	Финализирует выходное состояние, чтобы очистить все буферы. То есть все выходные буферы сбрасываются

Окончание табл. 19.7

Метод	Описание
`void write(byte буфер[])`	Записывает массив байтов в вызывающий поток
`void write(byte буфер[], int смещение, int колБайтов)`	Записывает диапазон *колБайтов* байт из массива *буфер* начиная с *буфер[смещение]*
`void write(int b)`	Записывает одиночный байт в вызывающий поток. Из аргумента *b* записывается только младший байт
`void writeObject(Object объект)`	Записывает объект *объект* в вызывающий поток

Класс ObjectOutputStream

Класс `ObjectOutputStream` расширяет класс `OutputStream` и реализует интерфейс `ObjectOutput`. Этот класс отвечает за запись объекта в поток. Конструктор его выглядит так.

```
ObjectOutputStream(OutputStream выходнойПоток) throws IOException
```

Аргумент *выходнойПоток* представляет собой выходной поток, в который могут быть записаны сериализуемые объекты. Закрытие объекта класса `ObjectOutputStream` приводит к закрытию также внутреннего потока, определенного аргументом *выходнойПоток*.

Несколько часто используемых методов класса перечислено в табл. 19.8. В случае ошибки все они передают исключение `IOException`. Присутствует также класс `PutField`, вложенный в класс `ObjectOutputStream`. Он обслуживает запись постоянных полей, и описание его применения выходит за рамки настоящей книги.

Таблица 19.8. Некоторые из наиболее часто используемых методов класса ObjectOutputStream

Метод	Описание
`void close()`	Закрывает вызывающий поток. Последующие попытки записи передадут исключение `IOException`. Внутренний поток тоже закрывается
`void flush()`	Финализирует выходное состояние, так что все буферы очищаются. То есть все выходные буферы сбрасываются
`void write(byte буфер[])`	Записывает массив байтов в вызывающий поток
`void write(byte буфер[], int смещение, int колБайтов)`	Записывает диапазон *колБайтов* байт из массива *буфер* начиная с *буфер[смещение]*
`void write(int b)`	Записывает одиночный байт в вызывающий поток. Из аргумента *b* записывается только младший байт
`void writeBoolean(boolean b)`	Записывает значение типа `boolean` в вызывающий поток
`void writeByte(int b)`	Записывает значение типа `byte` в вызывающий поток. Записываемый байт – младший из аргумента *b*
`void writeBytes(String строка)`	Записывает байты, составляющие строку `str`, в вызывающий поток
`void writeChar(int c)`	Записывает значение типа `char` в вызывающий поток

Окончание табл. 19.8

Метод	Описание
`void writeChars(String` *`строка`*`)`	Записывает символы, составляющие строку *`строка`*, в вызывающий поток
`void writeDouble(double` *`d`*`)`	Записывает значение типа `double` в вызывающий поток
`void writeFloat(float` *`f`*`)`	Записывает значение типа `float` в вызывающий поток
`void writeInt(int` *`i`*`)`	Записывает значение типа `int` в вызывающий поток
`void writeLong(long` *`l`*`)`	Записывает значение типа `long` в вызывающий поток
`final void writeObject(Object` *`объект`*`)`	Записывает объект *`объект`* в вызывающий поток
`void writeShort(int` *`i`*`)`	Записывает значение типа `short` в вызывающий поток

Интерфейс `ObjectInput`

Интерфейс `ObjectInput` расширяет интерфейсы `AutoCloseable` и `DataInput` и определяет методы, перечисленные в табл. 19.9. Он поддерживает сериализацию объектов. Особо стоит отметить метод `readObject()`. Он вызывается для десериализации объекта. В случае ошибок все эти методы передают исключение `IOException`. Метод `readObject()` также может передать исключение `ClassNotFoundException`.

Таблица 19.9. Методы, определенные в интерфейсе `ObjectInput`

Метод	Описание
`int available()`	Возвращает количество байтов, которые доступны во входном буфере в настоящий момент
`void close()`	Закрывает вызывающий поток. Последующие попытки чтения вызовут передачу исключения `IOException`. Внутренний поток тоже закрывается
`int read()`	Возвращает целочисленное представление следующего доступного байта ввода. При достижении конца файла возвращается значение `-1`
`int read(byte` *`буфер`*`[])`	Пытается прочитать до *`буфер.длина`* байт в *`буфер`*, начиная с *`буфер[смещение]`*, возвращая количество байтов, которые удалось прочитать. При достижении конца файла возвращается значение `-1`
`int read(byte` *`буфер`*`[], int` *`смещение`*`, int` *`колБайтов`*`)`	Пытается прочитать до *`колБайтов`* байт в *`буфер`*, начиная с *`буфер[смещение]`*, возвращая количество байтов, которые удалось прочитать. При достижении конца файла возвращается значение `-1`
`Object readObject()`	Читает объект из вызывающего потока
`Long skip(long numBytes)`	Игнорирует (т.е. пропускает) *`колБайтов`* байт вызывающего потока, возвращая количество действительно пропущенных байтов

Класс ObjectInputStream

Этот класс расширяет класс InputStream и реализует интерфейс ObjectInput. Класс ObjectInputStream отвечает за чтение объектов из потока. Ниже показан конструктор этого класса.

```
ObjectInputStream(InputStream входнойПоток) throws IOException
```

Аргумент *входнойПоток* — это входной поток, из которого должен быть прочитан сериализованный объект. Закрытие объекта класса ObjectInputStream приводит к закрытию также внутреннего потока, определенного аргументом *входнойПоток*.

Несколько часто используемых методов этого класса показано в табл. 19.10. В случае ошибок все они передают исключение IOException. Метод readObject() также может передать исключение ClassNotFoundException. Также в классе ObjectInputStream присутствует вложенный класс по имени GetField. Он обслуживает чтение постоянных полей, и описание его применения выходит за рамки настоящей книги.

Таблица 19.10. Часто используемые методы, определенные в классе ObjectInputStream

Метод	Описание
`int available()`	Возвращает количество байтов, доступных в данный момент во входном буфере
`void close()`	Закрывает вызывающий поток. Последующие попытки чтения вызовут передачу исключения IOException. Внутренний поток тоже закрывается
`int read()`	Возвращает целочисленное представление следующего доступного байта ввода. При достижении конца файла возвращается значение -1
`int read(byte буфер[], int смещение, int колБайтов)`	Пытается прочитать до *колБайтов* байт в *буфер*, начиная с *буфер[смещение]*, возвращая количество байтов, которые удалось прочитать. При достижении конца файла возвращается значение -1
`Boolean readBoolean()`	Читает и возвращает значение типа boolean из вызывающего потока
`byte readByte()`	Читает и возвращает значение типа byte из вызывающего потока
`char readChar()`	Читает и возвращает значение типа char из вызывающего потока
`double readDouble()`	Читает и возвращает значение типа double из вызывающего потока
`double readFloat()`	Читает и возвращает значение типа float из вызывающего потока
`void readFully(byte буфер[])`	Читает *буфер.длина* байт в *буфер*. Возвращает управление, только когда все байты прочитаны
`void readFully(byte буфер[], int смещение, int колБайтов)`	Читает *колБайтов* байт в *буфер*, начиная с *буфер[смещение]*. Возвращает управление, только когда прочитано *колБайтов* байт

Окончание табл. 19.10

Метод	Описание
`int readInt()`	Читает и возвращает значение типа `int` из вызывающего потока
`int readLong()`	Читает и возвращает значение типа `long` из вызывающего потока
`final Object readObject()`	Читает и возвращает объект из вызывающего потока
`short readShort()`	Читает и возвращает значение типа `short` из вызывающего потока
`int readUnsignedByte()`	Читает и возвращает значение типа `unsigned byte` из вызывающего потока
`int readUnsignedShort()`	Читает и возвращает значение типа `unsigned short` из вызывающего потока

Пример сериализации

В следующей программе показано, как использовать сериализацию и десериализацию объектов. Начинается она с создания экземпляра объекта `MyClass`. Этот объект имеет три переменные экземпляра типа `String`, `int` и `double`. Именно эту информацию мы хотим сохранять и восстанавливать.

В программе создается объект класса `FileOutputStream`, который ссылается на файл по имени `"serial"`, и для этого файлового потока создается объект класса `ObjectOutputStream`. Метод `writeObject()` этого объекта класса `ObjectOutputStream` используется затем для сериализации объекта. Объект выходного потока очищается и закрывается.

Далее создается объект класса `FileInputStream`, который ссылается на файл по имени `"serial"`, и для этого файлового потока создается объект класса `ObjectInputStream`. Метод `readObject()` класса `ObjectInputStream` используется для последующей десериализации объекта. После этого входной поток закрывается.

Обратите внимание на то, что объект `MyClass` определен с реализацией интерфейса `Serializable`. Если бы этого не было, передалось бы исключение `NotSerializableException`. Поэкспериментируйте с этой программой, объявляя некоторые переменные экземпляра `MyClass` как `transient`. Эти данные не будут сохраняться при сериализации.

```
// Демонстрация сериализации
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.

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("Исключение во время сериализации : " +
                               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("Исключение во время сериализации: " +
                               e);
            System.exit(0);
        }
    }
}

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;
    }
}
```

Эта программа демонстрирует идентичность переменных экземпляра объектов `object1` и `object2`. Вот ее вывод.

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

Преимущества потоков

Потоковый интерфейс ввода-вывода в Java предоставляет чистую абстракцию для сложных и зачастую обременительных задач. Композиция классов фильтрующих потоков позволяет динамически строить собственные настраиваемые потоковые интерфейсы, которые отвечают вашим требованиям к передаче данных. Программы Java, использующие эти абстрактные высокоуровневые классы — `InputStream`, `OutputStream`, `Reader` и `Writer`, — будут корректно функционировать в будущем, даже когда появятся новые усовершенствованные конкретные потоковые классы. Как вы увидите в главе 21, эта модель работает очень хорошо, когда мы переключаемся от набора потоков на основе файлов к сетевым потокам и потокам сокетов. И наконец, сериализация объектов играет важную роль в программах Java различных типов. Классы сериализации ввода-вывода Java обеспечивают переносимое решение этой непростой задачи.

ГЛАВА 20

Исследование NIO

Начиная с версии 1.4 язык Java предоставляет вторую систему ввода-вывода под названием NIO (сокращение от New I/O – *новый ввод-вывод*). Она поддерживает ориентированный на буферы канальный подход к операциям ввода-вывода. С появлением JDK 7 система NIO была существенно расширена, и теперь она оказывает улучшенную поддержку для средств обработки файлов и файловых систем. Фактически, изменения столь существенны, что ныне нередко используется термин *NIO.2*. Благодаря возможностям, предоставленным новыми файловыми классами NIO, ожидается, что система NIO станет еще более важной частью обработки файлов. Данная глава исследует некоторые основные характеристики системы NIO, включая ее новые возможности по обработке файлов.

Классы NIO

Рассматриваемые классы содержатся в следующих пакетах (табл. 20.1).

Таблица 20.1. Пакеты, содержащие классы NIO

Пакет	Назначение
`java.nio`	Пакет верхнего уровня в системе NIO. Инкапсулирует различные типы буферов, содержащих данные, с которыми работает система NIO
`java.nio.channels`	Поддерживает каналы, открывающие соединения ввода-вывода
`java.nio.channels.spi`	Поддерживает провайдеры служб для каналов
`java.nio.charset`	Инкапсулирует наборы символов. Также поддерживает работу кодировщиков и декодеров, преобразующих символы в байты и байты в символы соответственно
`java.nio.charset.spi`	Поддерживает провайдеры служб для наборов символов
`java.nio.file`	Обеспечивает поддержку для файлов. (Добавлено в JDK 7)
`java.nio.file.attribute`	Обеспечивает поддержку для атрибутов файлов. (Добавлено в JDK 7)
`java.nio.file.spi`	Поддержка провайдеров служб для файловых систем. (Добавлено в JDK 7)

Прежде чем приступить к рассмотрению системы NIO, следует понять, что эта система не предназначена для замены классов ввода-вывода, хранящихся в пакете `java.io`, о которых шла речь в главе 19. Наоборот, классы NIO дополняют стандартную систему ввода-вывода, предлагая альтернативный принцип, применение которого в некоторых случаях может быть предпочтительным.

Основы NIO

Система NIO построена на двух фундаментальных элементах: буферах и каналах. *Буфер* (buffer) хранит данные, а *канал* (channel) представляет открытое соединение с устройством ввода-вывода – с файлом или сокетом. В общем случае для использования системы NIO необходимо получить канал для устройства ввода-вывода и буфер для хранения данных. После этого вы будете работать с буфером, вводя или выводя данные по мере необходимости. В следующих разделах мы рассмотрим подробно буферы и каналы.

Буферы

Буферы определены в пакете `java.nio`. Все буферы являются подклассами класса `Buffer`, который определяет общие функциональные возможности, характерные для каждого буфера, – текущая позиция, предел и емкость. *Текущая позиция* (current position) представляет собой индекс в буфере, с которого в следующий раз начнется операция чтения или записи данных. Текущая позиция перемещается после выполнения большинства операций чтения или записи. *Предел* (limit) представляет собой значение индекса для вставки в последнюю допустимую ячейку буфера. *Емкость* определяет количество элементов, которые может хранить буфер. Предел зачастую равнозначен емкости буфера. Класс `Buffer` также поддерживает метку и сброс. Он определяет несколько методов, перечисленных в табл. 20.2.

Таблица 20.2. Методы, определенные в классе `Buffer`

Метод	Описание
`abstract Object array()`	Если вызывающий буфер поддерживается массивом, возвращает ссылку на массив. В противном случае передается исключение `UnsupportedOperationException`. Если массив доступен только для чтения, передается исключение `ReadOnlyBufferException`
`abstract int arrayOffset()`	Если вызывающий буфер поддерживается массивом, возвращает индекс первого элемента. В противном случае передается исключение `UnsupportedOperationException`. Если массив доступен только для чтения, передается исключение `ReadOnlyBufferException`
`final int capacity()`	Возвращает количество элементов, которые может хранить вызывающий буфер
`final Buffer clear()`	Очищает вызывающий буфер и возвращает ссылку на него
`final Buffer flip()`	Устанавливает предел вызывающего буфера по текущей позиции и сбрасывает текущую позицию до нуля. Возвращает ссылку на буфер
`abstract boolean hasArray()`	Возвращает значение `true`, если вызывающий буфер поддерживается массивом, доступным для чтения и записи. В противном случае возвращает значение `false`
`final boolean hasRemaining()`	Возвращает значение `true`, если существуют элементы, оставшиеся в вызывающем буфере. В противном случае возвращает значение `false`
`abstract boolean isDirect()`	Возвращает значение `true`, если вызывающий буфер прямой – другими словами, операции ввод-вывода выполняются с ним напрямую. В противном случае возвращает значение `false`
`abstract boolean isReadOnly()`	Возвращает значение `true`, если вызывающий буфер является буфером только для чтения. В противном случае возвращает значение `false`

Окончание табл. 20.2

Метод	Описание
`final int limit()`	Возвращает предел вызывающего буфера
`final Buffer limit(int n)`	Устанавливает предел вызывающего буфера в *n*. Возвращает ссылку на буфер
`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 буфера в *n*. Возвращает ссылку на буфер
`int remaining()`	Возвращает количество элементов, доступных до достижения предела. Другими словами, возвращает предел минус текущая позиция
`final Buffer reset()`	Сбрасывает текущую позицию вызывающего буфера в предварительно установленную метку. Возвращает ссылку на буфер
`final Buffer rewind()`	Устанавливает позицию вызывающего буфера в нуль. Возвращает ссылку на буфер

От класса `Buffer` происходят следующие специализированные классы буферов, тип хранимых данных которых можно определить по их именам.

`ByteBuffer`	`CharBuffer`	`DoubleBuffer`	`FloatBuffer`
`IntBuffer`	`LongBuffer`	`MappedByteBuffer`	`ShortBuffer`

Класс `MappedByteBuffer` является подклассом класса `ByteBuffer` и используется для отображения файла в виде буфера.

Каждый буфер предоставляют различные методы `get()` и `put()`, которые позволяют получать данные из буфера или вносить их в него. (Конечно, если буфер предназначен только для чтения, метод `put()` недоступен.) В табл. 20.3 представлены методы `get()` и `put()`, определенные классом `ByteBuffer`. Другие классы буферов имеют похожие методы. Все классы буферов также поддерживают методы, выполняющие различные операции с буфером. Например, с помощью метода `allocate()` можно вручную зарезервировать память под буфер. С помощью метода `wrap()` можно организовать массив внутри буфера. С помощью метода `slice()` можно создать подпоследовательность буфера.

Таблица 20.3. Методы `get()` и `put()`, определенные в классе `ByteBuffer`

Метод	Описание
`abstract byte get()`	Возвращает байт в текущей позиции
`ByteBuffer get(byte значения[])`	Копирует вызывающий буфер в массив, на который указывает параметр *значения*. Возвращает ссылку на буфер. Если в буфере не осталось *значения*.`length` элементов, передается исключение `BufferUnderflowException`
`ByteBuffer get(byte значения[], int начало, int число)`	Копирует *число* элементов из вызывающего буфера в массив, на который указывает параметр *значения*, начиная с индекса, заданного в параметре *начало*. Возвращает ссылку на буфер. Если в буфере не осталось *число* элементов, передается исключение `BufferUnderflowException`

Окончание табл. 20.3

Метод	Описание
`abstract byte get(int` *индекс*`)`	Возвращает из вызывающего буфера байт по индексу, заданному в параметре *индекс*
`abstract ByteBufferput(byte` *b*`)`	Копирует *b* в текущую позицию вызывающего буфера. Возвращает ссылку на буфер. Если буфер заполнен, передается исключение `BufferOverflowException`
`final ByteBufferput(byte` *значения*`[])`	Копирует все элементы массива *значения* в вызывающий буфер начиная с текущей позиции. Возвращает ссылку на буфер. Если буфер не может вместить все элементы, передается исключение `BufferOverflowException`
`ByteBuffer put(byte` *значения*`[], int` *начало*`, int` *число*`)`	Копирует *число* элементов из массива *значения*, начиная с позиции *начало*, в вызывающий буфер. Возвращает ссылку на буфер. Если буфер не может хранить все элементы, передается исключение `BufferOverflowException`
`ByteBufferput(ByteBuffer` *bb*`)`	Копирует элементы из *bb* в вызывающий буфер начиная с текущей позиции. Если буфер не может хранить все элементы, передается исключение `BufferOverflowException`. Возвращает ссылку на буфер
`abstract ByteBufferput(int` *индекс*`, byte` *b*`)`	Копирует *b* в позицию вызывающего буфера, заданную параметром *индекс*. Возвращает ссылку на буфер

Каналы

Каналы определены в пакете `java.io.channels`. *Канал* (channel) представляет открытое соединение с источником или назначением ввода-вывода. Классы каналов реализуют интерфейс `Channel`, расширяющий интерфейс `Closeable` и, начиная с JDK 7, интерфейс `AutoCloseable`. При реализации интерфейса `AutoCloseable` каналы могут управляться новым оператором JDK 7 *try-с-ресурсами*. При использовании в блоке оператора *try-с-ресурсами*, канал закрывается автоматически, когда он больше не нужен. (Более подробная информация об операторе *try-с-ресурсами* приведена в главе 13).

Один из способов получения канала подразумевает вызов метода `getChannel()` объекта, поддерживающего каналы. Например, метод `getChannel()` поддерживается следующими классами ввода-вывода.

`DatagramSocket`	`FileInputStream`	`FileOutputStream`
`RandomAccessFile`	`ServerSocket`	`Socket`

Специфический тип возвращаемого канала зависит от типа объекта, для которого вызывается метод `getChannel()`. Например, при вызове для объекта класса `FileInputStream`, `FileOutputStream` или `RandomAccessFile`, метод `getChannel()` возвращает канал типа `FileChannel`. При вызове для объекта класса `Socket`, этот метод возвращает канал типа `SocketChannel`.

Еще один способ получения канала подразумевает использование одного из статических методов, определенных классом `Files`, который был добавлен в комплекте JDK 7. Например, используя класс `Files`, вы можете получить байтовый канал при вызове метода `newByteChannel()`. Он возвращает канал типа

SeekableByteChannel, являющегося интерфейсом, реализованным классом FileChannel. (Подробнее класс Files рассматривается далее в этой главе.)

Каналы вроде FileChannel и SocketChannel поддерживают различные методы read() и write(), которые позволяют выполнять операции ввода-вывода через канал. Например, в табл. 20.4 показано несколько методов read() и write(), определенных для класса FileChannel.

Таблица 20.4. Методы read() и write(), определенные в классе FileChannel

Метод	Описание
`abstract int read(ByteBuffer bb) throws IOException`	Считывает байты из вызывающего канала в *bb* до тех пор, пока буфер не будет заполнен или пока не закончатся входные данные. Возвращает количество прочитанных байтов
`abstract int read(ByteBuffer bb, long начало) throws IOException`	Начиная с позиции, определяемой при помощи параметра *начало*, считывает байты из вызывающего канала в *bb* до тех пор, пока буфер не будет заполнен или пока не закончатся входные данные. Текущая позиция не изменяется. Возвращает количество прочитанных байтов или значение -1, если начальная позиция (параметр *начало*) окажется за границами файла
`abstract int write(ByteBuffer bb) throws IOException`	Записывает содержимое байтового буфера в вызывающий канал начиная с текущей позиции. Возвращает количество записанных байтов
`abstract int write(ByteBuffer bb, long начало) throws IOException`	Начиная с позиции файла, определяемой при помощи параметра *начало*, записывает содержимое байтового буфера в вызывающий канал. Текущая позиция не изменяется. Возвращает количество записанных байтов

Все каналы поддерживают дополнительные методы, предоставляющие доступ к каналу и позволяющие управлять им. Например, канал класса FileChannel поддерживает методы для получения и установки текущей позиции, передачи информации между файловыми каналами, получения текущего размера канала и его блокировки. Начиная с JDK 7 класс FileChannel предоставляет статический метод open(), который открывает файл и возвращает канал для него. Это обеспечивает другой способ получения канала. Класс FileChannel предоставляет также метод map(), с помощью которого можно соотнести файл с буфером.

Наборы символов и селекторы

Система NIO использует наборы символов и селекторы. *Набор символов* (charset) определяет способ соотнесения байтов с символами. С помощью *кодировщика* (encoder) можно зашифровать последовательность символов в виде байтов. Процесс расшифровки производится с помощью *декодера* (decoder). Наборы символов, кодировщики и декодеры поддерживаются классами, определенными в пакете java.nio.charset. Поскольку кодировщики и декодеры предлагаются по умолчанию, работать с наборами символов явным образом вам придется очень редко.

Селектор (selector) обеспечивает возможность многоканального ввода-вывода на основе ключей без применения блокировки. Другими словами, с помощью селекторов можно выполнять операции ввода-вывода при помощи нескольких каналов. Селекторы поддерживаются классами, определенными в пакете java.nio.channels. Селекторы чаще всего применяются в каналах на основе сокетов.

В этой главе мы не будем использовать наборы символов или селекторы, однако их применение может оказаться очень полезным в ряде приложений.

Дополнения, внесенные в NIO (комплект JDK 7)

Начиная с JDK 7 система NIO была существенно расширена и усовершенствована. Кроме поддержки оператора `try-с-ресурсами` (который обеспечивает автоматическое управление ресурсами), усовершенствования включают 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 для потокового ввода-вывода. Дополнения существенно расширили способы применения NIO, особенно с файлами. В следующих разделах описаны некоторые ключевые добавления.

Интерфейс `Path`

Возможно, одним из наиболее важных дополнений к системе NIO является интерфейс `Path`, поскольку он инкапсулирует путь к файлу. Как вы увидите, интерфейс `Path` — это связующий элемент, объединяющий большинство новых файловых средств NIO.2. Он описывает расположение файла в пределах структуры каталогов. Интерфейс `Path` находится в пакете `java.nio.file` и наследует интерфейсы `Watchable`, `Iterable<Path>` и `Comparable<Path>`. Интерфейс `Watchable` описывает объект, который может быть наблюдаем и изменяем. Он также был добавлен в JDK 7. Интерфейсы `Iterable` и `Comparable` были описаны ранее в этой книге.

Интерфейс `Path` объявляет множество методов для работы с путями. Некоторые из них приведены в табл. 20.5. Обратите особое внимание на метод `getName()`. Он используется для получения элемента пути. Для этого он применяет индекс. Нулевому значению индекса соответствует ближайшая к корню часть пути, являющаяся крайним левым его элементом. Последующие индексы определяют элементы вправо от корня. Количество элементов в пути может быть получено при вызове метода `getNameCount()`. Если вы хотите получить строковое представление всего пути, просто вызовите метод `toString()`. Обратите внимание, что вы можете распознать относительный и абсолютный путь при помощи метода `resolve()`.

Таблица 20.5. Некоторые методы, определенные в интерфейсе `Path`

Метод	Описание
`boolean endsWith(String` *`путь`*`)`	Возвращает значение `true`, если вызывающий объект интерфейса `Path` заканчивается путем, указанным аргументом *`путь`*. В противном случае возвращает значение `false`
`boolean endsWith(Path` *`путь`*`)`	Возвращает значение `true`, если вызывающий объект интерфейса `Path` заканчивается путем, указанным аргументом *`путь`*. В противном случае возвращает значение `false`
`Path getFileName()`	Возвращает имя файла, связанное с вызывающим объектом интерфейса `Path`
`Path getName(int` *`индекс`*`)`	Возвращает объект интерфейса `Path`, содержащий имя элемента пути, заданного аргументом *`индекс`* в пределах вызывающего объекта. Крайний левый элемент имеет индекс 0. Это элемент, ближайший к корню. Крайний правый элемент имеет индекс `getNameCount() - 1`
`int getNameCount()`	Возвращает количество элементов (кроме корневого) в вызывающем объекте интерфейса `Path`

Окончание табл. 20.5

Метод	Описание
`Path getParent()`	Возвращает объект интерфейса `Path`, содержит весь путь, за исключением имени файла, определенного вызывающим объектом интерфейса `Path`
`Path getRoot()`	Возвращает корневой каталог вызывающего объекта интерфейса `Path`
`boolean isAbsolute()`	Возвращает значение `true`, если вызывающий объект интерфейса `Path` является абсолютным. В противном случае возвращает значение `false`
`Path resolve(Path` *путь*`)`	Если *путь* является абсолютным, возвращается *путь*. В противном случае, если *путь* не содержит корневой каталог, *путь* предваряется корневым каталогом, определенным вызывающим объектом интерфейса `Path`, а результат возвращается. Если *путь* пуст, возвращается вызывающий объект `Path`. В противном случае поведение не определено
`Path resolve(String` *путь*`)`	Если *путь* является абсолютным, возвращается *путь*. В противном случае, если *путь* не содержит корневой каталог, *путь* предваряется корневым каталогом, определенным вызывающим объектом `Path`, а результат возвращается. Если *путь* пуст, возвращается вызывающий объект интерфейса `Path`. В противном случае поведение не определено
`boolean startsWith(String` *путь*`)`	Возвращает значение `true`, если вызывающий объект интерфейса `Path` начинается с пути, определенного аргументом *путь*. В противном случае возвращает значение `false`
`boolean startsWith(Path` *путь*`)`	Возвращает значение `true`, если вызывающий объект интерфейса `Path` начинается с пути, определенного аргументом *путь*. В противном случае возвращает значение `false`
`Path toAbsolutePath()`	Возвращает вызывающий объект интерфейса `Path` как абсолютный путь
`String toString()`	Возвращает строковое представление вызывающего объекта интерфейса `Path`

Еще один момент: при обновлении устаревшего кода, который использует класс `File`, определенный в пакете `java.io`, можно преобразовать экземпляр класса `File` в экземпляр интерфейса `Path`, вызвав метод `toPath()` для объекта класса `File`. Этот метод был добавлен к классу `File` в JDK 7. Кроме того, можно получить экземпляр класса `File`, вызвав метод `toFile()`, определенный в интерфейсе `Path`.

Класс `Files`

Большинство действий, которые вы выполняете с файлами, представлены статическими методами класса `Files`. Файл, с которым осуществляются действия, задает его путь. Таким образом, методы класса `Files` используют объект интерфейса `Path`, чтобы определить используемый файл. Класс `Files` содержит широкое разнообразие функциональных возможностей. Например, у него есть методы, позволяющие открывать или создавать файл, расположенный по определенному пути. Вы можете получить информацию об объекте интерфейса `Path` – исполняемый ли это файл, скрытый или только для чтения. Класс `Files` также предостав-

ляет методы, позволяющие вам копировать или перемещать файлы. Некоторые его методы представлены в табл. 20.6. Кроме исключения `IOException`, возможна передача и некоторых других исключений.

Таблица 20.6. Некоторые его методы, определенные в классе `Files`

Метод	Описание
`static Path copy(Path` *ист*`, Path` *назн*`, CopyOption ... ` *как*`) throws IOException`	Копирует файл, определенный параметром *ист*, в позицию, указанную параметром *назн*. Параметр *как* определяет, как будет осуществляться копирование
`static Path createDirectory(Path` *путь*`, FileAttribute<?> ... ` *атрибуты*`) throws IOException`	Создает каталог, путь которого определяется параметром *путь*. Атрибуты каталога определяет параметр *атрибуты*
`static Path createFile(Path` *путь*`, FileAttribute<?> ... ` *атрибуты*`) throws IOException`	Создает файл, путь к которому определяет параметр *путь*. Атрибуты файла указаны в параметре *атрибуты*
`static void delete(Path` *путь*`) throws IOException`	Удаляет файл, путь к которому определяет параметр *путь*
`static boolean exists(Path` *путь*`, LinkOptions ... ` *парам*`)`	Возвращает значение `true`, если файл, определенный параметром *путь*, существует, и значение `false` — в противном случае. Если параметр *парам* не определен, то используются символические ссылки. Чтобы предотвратить следование по символическим ссылкам, передайте в параметре *парам* значение `NOFOLLOW_LINKS`
`static boolean isDirectory(Path` *путь*`, LinkOptions ... ` *парам*`)`	Возвращает значение `true`, если параметр *путь* определяет каталог, и значение `false` — в противном случае. Если параметр *парам* не определен, то используются символические ссылки. Чтобы предотвратить следование по символическим ссылкам, передайте в параметре *парам* значение `NOFOLLOW_LINKS`
`static boolean isExecutable(Path` *путь*`)`	Возвращает значение `true`, если файл, определенный параметром *путь*, является исполняемым, и значение `false` — в противном случае
`static boolean isHidden(Path` *путь*`) throws IOException`	Возвращает значение `true`, если файл, определенный параметром *путь*, является скрытым, и значение `false` — в противном случае
`static boolean isReadable(Path` *путь*`)`	Возвращает значение `true`, если файл, определенный параметром *путь*, допускает чтение, и значение `false` — в противном случае
`static boolean isRegularFile(Path` *путь*`, LinkOptions ... ` *парам*`)`	Возвращает значение `true`, если параметр *путь* определяет файл, и значение `false` — в противном случае. Если параметр *парам* не определен, то используются символические ссылки. Чтобы предотвратить следование по символическим ссылкам, передайте в параметре *парам* значение `NOFOLLOW_LINKS`
`static boolean isWritable(Path` *путь*`)`	Возвращает значение `true`, если файл, определенный параметром *путь*, допускает запись, и значение `false` — в противном случае

Окончание табл. 20.6

Метод	Описание
`static Path move(Path` *ист*`, Path` *назн*`, CopyOption ... ` *как*`) throws IOException`	Перемещает файл, определенный параметром *ист*, в позицию, указанную параметром *назн*. Параметр *как* определяет, как будет осуществляться копирование
`static SeekableByteChannel newByteChannel(Path` *путь*`, OpenOption ... ` *как*`) throws IOException`	Открывает файл, определенный параметром *путь*, как указано параметром *как*. Возвращает байтовый канал интерфейса `SeekableByteChannel` для файла. Текущая позиция этого байтового канала может быть изменена. Интерфейс `SeekableByteChannel` реализован классом `FileChannel`
`static DirectoryStream<Path> newDirectoryStream(Path` *путь*`) throws IOException`	Открывает каталог, определенный параметром *путь*. Возвращает объект `DirectoryStream`, связанный с каталогом
`static InputStream newInputStream(Path` *путь*`, OpenOption ... ` *как*`) throws IOException`	Открывает файл, определенный параметром *путь*, так, как указано параметром *как*. Возвращает объект класса `InputStream`, связанный с файлом
`static OutputStream newOutputStream(Path` *путь*`, OpenOption ... ` *как*`) throws IOException`	Открывает файл, определенный вызывающим объектом, как указано параметром *как*. Возвращает объект класса `OutputStream`, связанный с файлом
`static boolean notExists(Path` *путь*`, LinkOption ... ` *парам*`)`	Возвращает значение `true`, если файл, определенный параметром *путь*, *не существует*, и значение `false` — в противном случае. Если параметр *парам* не определен, то используются символические ссылки. Чтобы предотвратить следование по символическим ссылкам, передайте в параметре *парам* значение `NOFOLLOW_LINKS`
`static <A extends BasicFileAttributes> A readAttributes(Path` *путь*`, Class<A> ` *типАтрибута*`, LinkOption ... ` *парам*`) throws IOException`	Получает атрибуты, связанные с файлом, определенным параметром *путь*. Тип передаваемых атрибутов определяется параметром *типАтрибута*. Если параметр *парам* не определен, то используются символические ссылки. Чтобы предотвратить следование по символическим ссылкам, передайте в параметре *парам* значение `NOFOLLOW_LINKS`
`static long size(Path` *путь*`) throws IOException`	Возвращает размер файла, определенного параметром *путь*

Обратите внимание на то, что некоторые методы в табл. 20.6 получают аргумент типа `OpenOption`. Это интерфейс, описывающий способ открытия файла. Его реализует класс `StandardOpenOption`, определяющий перечисление, значения которого представлены в табл. 20.7.

Таблица 20.7. Стандартные параметры открытия

Значение	Смысл
`APPEND`	Задает запись вывода в конец файла
`CREATE`	Задает создание файла, если он еще не существует
`CREATE_NEW`	Задает создание файла, только если он еще не существует
`DELETE_ON_CLOSE`	Задает удаление файла, когда он закрывается

Окончание табл. 20.7

Значение	Смысл
DSYNC	Задает немедленную запись вносимых изменений в физический файл. Обычно, для повышения производительности, изменения в файле буферизируются файловой системой и записываются только при необходимости
READ	Открыть файл для операций ввода
SPARSE	Указывает файловой системе, что файл разрежен, а значит, он не может быть полностью заполнен данными. Если файловая система не поддерживает разреженные файлы, это значение параметра игнорируется
SYNC	Задает немедленную запись вносимых изменений файла или его метаданных в физический файл. Обычно, для повышения производительности, изменения в файле буферизируются файловой системой и записываются только при необходимости
TRUNCATE_EXISTING	Задает усечение существующего ранее файла при открытии для вывода, уменьшая его длину до нуля
WRITE	Открывает файл для операций вывода

Класс Paths

Поскольку Path — это интерфейс, а не класс, вы не можете создать его экземпляр непосредственно, с помощью конструктора. Вместо этого вы получаете объект пути, вызывая метод, который возвращает его. Как правило, для этого используется метод get(), определяемый классом Paths. Существует две формы метода get(). Одна, используемая в этой главе, имеет такой вид:

```
static Path get(String имяпути, String ... части)
```

Он возвращает объект, инкапсулирующий определенный путь. Путь может быть задан двумя способами. Если параметр *части* не используется, то путь должен быть полностью определен параметром *имяпути*. В качестве альтернативы вы можете передать путь по частям, с первой частью в параметре *имяпути* и в последующих частях, определенных в аргументе переменной длины параметра *части*. В любом случае, если определенный путь синтаксически недопустим, метод get() передаст исключение InvalidPathException.

Вторая форма метода get() создает путь из идентификатора URI и имеет такой вид.

```
static Path get(URI uri)
```

Возвращается путь, соответствующий значению параметра *uri*.

Следует понять, что создание пути к файлу не открывает и не создает файл. В результате просто создается объект, который инкапсулирует путь к каталогу файла.

Интерфейсы атрибутов файла

С файлами связан ряд атрибутов — время создания файла, время его последней модификации, является ли файл каталогом и его размер. Система NIO организует файловые атрибуты в несколько разных интерфейсов. Атрибуты представлены иерархией интерфейсов, определенных в пакете java.nio.file.attribute. Верховным является интерфейс BasicFileAttributes. Он инкапсулирует набор атрибутов,

которые обычно используются большинством файловых систем. Методы, определенные в интерфейсе `BasicFileAttributes`, представлены в табл. 20.8.

Таблица 20.8. Методы, определенные интерфейсом `BasicFileAttributes`

Метод	Описание
`FileTime creationTime()`	Возвращает время создания файла. Если файловая система не поддерживает время создания, то возвращается значение, зависящее от реализации
`Object fileKey()`	Возвращает ключ файла. Если это не поддерживается, возвращается значение `null`
`boolean isDirectory()`	Возвращает значение `true`, если файл представляет собой каталог
`boolean isOther()`	Возвращает значение `true`, если файл является не файлом, а символической ссылкой или каталогом
`boolean isRegularFile()`	Возвращает значение `true`, если файл является обычным файлом, а не каталогом или символической ссылкой
`boolean isSymbolicLink()`	Возвращает значение `true`, если файл – символическая ссылка
`FileTime lastAccessTime()`	Возвращает время последнего обращения к файлу. Если файловая система не поддерживает время последнего обращения, то возвращается значение, зависящее от реализации
`FileTime lastModifiedTime()`	Возвращает время последней модификации файла. Если файловая система не поддерживает время последней модификации, то возвращается значение, зависящее от реализации
`long size()`	Возвращает размер файла

От интерфейса `BasicFileAttributes` происходят два интерфейса: `DosFileAttributes` и `PosixFileAttributes`. Интерфейс `DosFileAttributes` описывает атрибуты, связанные с файловой системой FAT, которые ранее были представлены файловой системой DOS. Здесь определены методы, указанные в табл. 20.9.

Таблица 20.9. Методы, определенные интерфейсом `DosFileAttributes`

Метод	Описание
`boolean isArchive()`	Возвращает значение `true`, если файл помечен как архивный, и значение `false` – в противном случае
`boolean isHidden()`	Возвращает значение `true`, если файл помечен как скрытый, и значение `false` – в противном случае
`boolean isReadOnly()`	Возвращает значение `true`, если файл помечен как только для чтения, и значение `false` – в противном случае
`boolean isSystem()`	Возвращает значение `true`, если файл помечается как системный, и значение `false` – в противном случае

Интерфейс `PosixFileAttributes` инкапсулирует атрибуты, определенные по стандартам POSIX (Portable Operating System Interface – переносимый интерфейс операционных систем). Здесь определены методы, представленные в табл. 20.10.

Таблица 20.10. Методы, определенные интерфейсом `PosixFileAttributes`

Метод	Описание
`GroupPrincipal group()`	Возвращает группу владельца файла
`UserPrincipal owner()`	Возвращает владельца файла
`Set<PosixFilePermission> permissions()`	Возвращает права файла

Есть разные способы доступа к атрибутам файла. Можно получить объект, который инкапсулирует атрибуты файла, вызвав статический метод `readAttributes()`, определенный в классе `Files`. Вот одна из его форм.

```
static <A extends BasicFileAttributes> A readAttributes(Path путь,
Class<A> типАтрибута, LinkOption... парам) throws IOException
```

Этот метод возвращает ссылку на объект, который определяет атрибуты, связанные с файлом, указанным параметром *путь*. Специфический тип атрибутов определяется как объект класса `Class` в параметре *типАтрибута*. Например, чтобы получить основные атрибуты файла, передайте в параметре *типАтрибута* значение `BasicFileAttributes.class`. Для атрибутов DOS используйте значение `DosFileAttributes.class`, а для атрибутов POSIX — значение `PosixFileAttributes.class`. Необязательные параметры ссылки передаются в параметре *парам*. Если он не определен, то используются символические ссылки. Метод возвращает ссылку на требуемый атрибут. Если требуемый тип атрибута недоступен, передается исключение `UnsupportedOperationException`. Используя возвращенный объект, вы можете обратиться к атрибутам файла.

Еще один способ доступа к атрибутам файла подразумевает вызов метода `getFileAttributeView()`, определенного в классе `Files`. Система NIO определяет несколько интерфейсов представления атрибутов, включая `AttributeView`, `BasicFileAttributeView`, `DosFileAttributeView` и `PosixFileAttributeView`, кроме прочих. Хотя мы не будем использовать представления атрибута в этой главе, это средство может оказаться полезным в некоторых ситуациях.

В некоторых случаях вам не нужно будет использовать сами интерфейсы атрибута файла, поскольку класс `Files` предоставляет удобные статические методы, позволяющие обращаться к некоторым атрибутам. Например, класс `Files` включает такие методы, как `isHidden()` и `isWritable()`.

Важно понять, что не все файловые системы поддерживают все возможные атрибуты. Например, атрибуты файла DOS относятся к файловой системе FAT, хотя сначала они были определены в файловой системе DOS. Те атрибуты, которые относятся к широкому разнообразию файловых систем, описаны в интерфейсе `BasicFileAttributes`. Поэтому в примерах данной главы используются эти атрибуты.

Классы `FileSystem`, `FileSystems` и `FileStore`

Комплект JDK 7 облегчает доступ к файловой системе, предоставляя в пакете `java.nio.file` дополнительные классы `FileSystem` и `FileSystems`. Фактически, используя метод `newFileSystem()`, определенный в классе `FileSystems`, можно даже получить новую файловую систему. Класс `FileStore` инкапсулирует систему хранения файла. Хотя эти классы не используются в данной главе непосредственно, вы можете найти их полезными в собственных приложениях.

Использование системы NIO

Этот раздел иллюстрирует применение системы NIO для множества задач. Однако вначале следует подчеркнуть, что с выпуском JDK 7 была существенно расширена как сама система NIO, так и область ее применения. Как упоминалось, усовершенствованная версия иногда называется NIO.2. В результате появления системы NIO.2 был изменен способ написания кода на базе системы NIO, а также расширен диапазон задач, к которым она может быть применена. В большей части материала и примеров этой главы используются средства системы NIO.2, поэтому для работы необходим комплект JDK версии 7 или выше. Однако в конце главы дано краткое описание кода, использованного до JDK 7, — в помощь тем программистам, которые работают с системой до JDK 7 или поддерживают устаревший код.

Помните! Для большинства примеров этой главы требуется комплект JDK 7.

Ранее главной задачей системы NIO был канальный ввод-вывод, и это все еще остается ее важнейшей областью применения. Однако теперь вы можете использовать систему NIO и для потокового ввода-вывода, и для выполнения операций файловой системы. В результате обсуждение использования системы NIO делится на три части:

- для канального ввода-вывода;
- для потокового ввода-вывода;
- для операций файловой системы.

Поскольку наиболее распространенным устройством ввода-вывода является диск с файлами, далее в этой главе они и используются. Поскольку все канальные операции с файлами имеют байтовую основу, типом буферов, которые нам предстоит использовать, будет `ByteBuffer`.

Прежде чем вы сможете открыть файл для доступа при помощи системы NIO, необходимо получить объект интерфейса `Path`, который описывает файл. Один из способов сделать это — вызов метода фабрики `Paths.get()`, который был описан ранее. В приведенных здесь примерах используется такая форма метода `get()`.

```
static Path get(String имяпути, String ... части)
```

Напомню, что путь может быть определен двумя способами. Он может быть передан по частям — в параметре *имяпути* и последующих частях, определенных аргументами переменной длины параметра *части*. В качестве альтернативы весь путь может быть определен в параметре *имяпути*, а параметр *части* не используется. Именно этот подход используется в примерах.

Использование системы NIO для канального ввода-вывода

Важнейшей областью использования системы NIO является получение доступа к файлу через каналы и буферы. Следующие разделы демонстрируют несколько способов использования канала для чтения файла и записи в него.

Чтение файла через канал

Есть несколько способов чтения данных из файла с использованием канала. Вероятно, наиболее распространенный способ — создание буферов вручную и по-

следующее явное выполнение операций чтения, которые загружают эти буферы данными из файла. Именно с этого подхода и начнем.

Прежде чем можно будет читать из файла, его необходимо открыть. Для этого сначала создайте объект интерфейса `Path`, который описывает файл. Затем используйте его, чтобы открыть файл. Есть разные способы открытия файла, в зависимости от того, как он будет использоваться. В этом примере файл будет открыт для байтового ввода при помощи явных операций ввода. Поэтому данный пример откроет файл и установит канал доступа к нему при вызове метода `Files.newByteChannel()`. Метод `newByteChannel()` имеет следующую общую форму.

```
static SeekableByteChannel newByteChannel(Path путь, OpenOption ... как)
throws IOException
```

Он возвращает объект интерфейса `SeekableByteChannel`, инкапсулирующий канал для файловых операций. Объект интерфейса `Path`, который описывает файл, передается в параметре *путь*. Параметр *как* определяет, как файл будет открыт. Поскольку это параметр для аргумента переменной длины, вы можете определить любое количество аргументов, отделяемых запятыми. (Допустимые значения обсуждались ранее и приведены в табл. 20.7.) Если никаких аргументов не определено, файл открывается для операций ввода. Интерфейс `SeekableByteChannel` описывает канал, применяемый для файловых операций. Он реализуется классом `FileChannel`. Когда используется файловая система, заданная по умолчанию, возвращенный объект может быть приведен к классу `FileChannel`. По завершении работы с каналом следует закрыть его. Поскольку все каналы, включая класс `FileChannel`, реализуют интерфейс `AutoCloseable`, вы можете использовать оператор *try-с-ресурсами* для автоматического закрытия файла, вместо явного вызова метода `close()`. Именно этот подход и используется в примерах.

Затем следует получить буферы, которые будут использоваться каналом либо при заключении в оболочку существующего массива, либо при динамическом резервировании буферов. В примерах используется резервирование, но выбор за вами. Поскольку файловые каналы работают с байтовыми буферами, для их получения мы будем использовать метод `allocate()`, определенный в классе `ByteBuffer`. Его общая форма такова.

```
static ByteBuffer allocate(int емкость)
```

Здесь параметр *емкость* определяет емкость буферов. Возвращается ссылка на буферы.

После создания буферов вы вызываете метод `read()` объекта канала, передавая ссылку на буферы. Ниже представлена версия метода `read()`, которую мы будем использовать далее.

```
int read(ByteBuffer буфер) throws IOException
```

При каждом вызове метод `read()` заполняет данными из файла буферы, определенные параметром *буфер*. Чтение осуществляется последовательно, значит, каждый вызов метода `read()` читает из файла в буфер следующую последовательность байтов. Метод `read()` возвращает количество фактически прочитанных байтов. При попытке чтения после конца файла, возвращается значение `-1`.

Следующая программа демонстрирует изложенное выше на практике, при чтении файла `test.txt` через канал с использованием явных операций ввода.

```
// Использование канала ввода-вывод для чтения файла. Требует JDK 7.

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 = Paths.get("test.txt");
        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
            return;
        }

        // Затем получить канал к этому файлу в пределах
        // блока try-с-ресурсами.
        try (SeekableByteChannel fChan = Files.newByteChannel(filepath))
        {

            // Зарезервировать буфер.
            ByteBuffer mBuf = ByteBuffer.allocate(128);

            do {
                // Читать в буфер.
                count = fChan.read(mBuf);

                // Остановиться при достижении конца файла.
                if(count != -1) {

                    // Подготовить буфер для чтения.
                    mBuf.rewind();

                    // Читать байты в буфер и отображать 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("I/O Error " + e);
        }
    }
```

Вот как работает эта программа. Сначала создается объект интерфейса `Path`, содержащий относительный путь к файлу по имени `test.txt`. Ссылка на этот объект присваивается переменной `filepath`. Затем, при вызове метода `newByteChannel()`, создается канал, связанный с файлом, указанным переданной переменной `filepath`. Поскольку никаких параметров не определено, файл открывается для чтения. Обратите внимание, что этот канал — объект, управляемый оператором *try-с-ресурсами*. Таким образом, канал автоматически закрывается в конце блока. Затем программа вызывает метод `allocate()` класса `ByteBuffer`, чтобы зарезервировать буфер для хранения содержимого файла во время чтения. Ссылка на этот буфер хранится в объекте `mBuf`. Затем, при вызове метода `read()`, содержимое файла читается по одному буферу за раз в объект `mBuf`. Количество прочитанных байтов сохраняется в переменной `count`. Затем буфер возобновля-

ется при вызове метода rewind(). Этот вызов необходим, поскольку после вызова метода read() текущая позиция находится в конце буфера. Ее следует возвратить в начало буфера, чтобы при вызове метода get() байты могли быть прочитаны в объект mBuf. (Напомню, что метод get() определен в классе ByteBuffer.) Поскольку объект mBuf — это байтовый буфер, значения, возвращенные методом get(), являются байтами. Они приводятся к типу char, таким образом, файл может быть отображен как текст. (В качестве альтернативы можно создать буфер, который перекодирует байты в символы, а затем прочитает этот буфер.) При достижении конца файла метод read() возвращает значение -1. Когда это происходит, программа заканчивает работу и канал автоматически закрывается.

Обратите внимание на интересный момент: программа получает объект интерфейса Path в пределах одного блока try, а затем использует его в другом блоке try для получения канала, связанного с этим путем, и работы с ним. Хотя ничего неправильного, по существу, в этом подходе нет, во многих случаях он может быть упрощен, чтобы использовался только один блок try. В этом случае вызовы методов Paths.get() и newByteChannel() объединяются. Вот, например, переделанная версия программы, которая использует этот подход.

```
// Более компактный способ открытия канала. Требует JDK 7.

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 по пути, возвращенному
        // методом Paths.get().
        // Переменная filepath больше не нужна.
        try (SeekableByteChannel fChan =
             Files.newByteChannel(Paths.get("test.txt")) )
        {
            // Зарезервировать буфер.
            ByteBuffer mBuf = ByteBuffer.allocate(128);

            do {
                // Читать в буфер.
                count = fChan.read(mBuf);

                // Остановиться при достижении конца файла.
                if(count != -1) {

                    // Подготовить буфер для чтения.
                    mBuf.rewind();

                    // Читать байты в буфер и отображать 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("Path Error " + e);
        } catch (IOException e) {
```

```
            System.out.println("I/O Error " + e);
        }
    }
}
```

В этой версии нет необходимости в переменной `filepath`, и оба исключения обрабатываются тем же оператором `try`. Поскольку этот подход компактнее, он используется в остальных примерах данной главы. Конечно, в собственном коде вы можете столкнуться с ситуацией, когда создание объекта интерфейса `Path` должно быть отделено от получения канала. В таких случаях применяется предыдущий подход.

Другой способ чтения файла подразумевает соотнесение его с буфером. Преимущество заключается в том, что буфер автоматически получает содержимое файла. Никаких явных операций чтения не нужно. Сопоставление и чтение содержимого файла осуществляются в ходе следующей общей процедуры. Сначала получите объект интерфейса `Path`, инкапсулирующий файл, как описано ранее. Затем получите канал к этому файлу, передав при вызове метода `Files.newByteChannel()` объект интерфейса `Path` и приведя тип возвращенного объекта к типу `FileChannel`. Как упоминалось, метод `newByteChannel()` возвращает объект интерфейса `SeekableByteChannel`. При использовании заданной по умолчанию файловой системы этот объект может быть приведен к типу класса `FileChannel`. Затем сопоставьте канал с буфером, вызвав метод `map()` для канала. Метод `map()` определен в классе `FileChannel`. Вот почему приведение к типу `FileChannel` необходимо. Вот синтаксис метода `map()`.

```
MappedByteBuffer map(FileChannel.MapMode как, long позиция, long размер)
throws IOException
```

Метод `map()` сопоставляет данные в файле с буфером в памяти. Значение параметра *как* определяет, какие операции разрешены. Для него допустимы следующие значения.

`MapMode.READ_ONLY`	`MapMode.READ_WRITE`	`MapMode.PRIVATE`

Чтобы читать файл, используйте значение `MapMode.READ_ONLY`. Чтобы читать и записывать файл, используйте значение `MapMode.READ_WRITE`. Значение `MapMode.PRIVATE` приводит к созданию закрытой копии файла, чтобы внесенные в буфере изменения не повлияли на основной файл. Позиция начала сопоставления в пределах файла определяется параметром *позиция*, а количество сопоставляемых байтов — параметром *размер*. Ссылка на этот буфер возвращается как объект класса `MappedByteBuffer`, который является производным от класса `ByteBuffer`. Как только файл сопоставлен с буфером, вы можете читать файл из этого буфера. Рассмотрим пример, иллюстрирующий этот подход.

```
// Использование сопоставления для чтения файла. Требует JDK 7.

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 для файла в пределах блока try-с-ресурсами.
        try ( FileChannel fChan =
            (FileChannel) Files.newByteChannel(Paths.get("test.txt")) )
        {

            // Получить размер файла.
```

```
            long fSize = fChan.size();

            // Теперь сопоставить файл с буфером.
            MappedByteBuffer mBuf =
                 fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

            // Читать и отображать байты из буфера.
            for(int i=0; i < fSize; i++)
                System.out.print((char)mBuf.get());

            System.out.println();

        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch (IOException e) {
            System.out.println("I/O Error " + e);
        }
    }
}
```

В программе сначала создается путь к файлу, а затем открывается файл при помощи метода `newByteChannel()`. Канал приводится к типу `FileChannel` и сохраняется в объекте `fChan`. Затем, в результате вызова метода `size()` для канала, выясняется размер файла. Далее весь файл сопоставляется с областью в памяти при вызове метода `map()` для объекта `fChan`, а ссылка на буфер сохраняется в объекте `mBuf`. Обратите внимание, что объект `mBuf` объявляется как ссылка на объект класса `MappedByteBuffer`. Байты из объекта `mBuf` читаются непосредственно методом `get()`.

Запись в файл через канал

Как и в случае чтения из файла, есть несколько способов записи данных в файл с использованием канала. Начнем с одного из наиболее распространенных. При этом подходе вы вручную резервируете буфер, записываете в него данные, а затем выполняете явную операцию записи этих данных в файл.

Прежде чем вы сможете записать файл, его следует открыть. Для этого получите сначала объект интерфейса `Path`, описывающий файл, а затем используйте этот путь, чтобы открыть файл. В данном примере файл будет открыт для байтового вывода с помощью явных операций вывода. Поэтому данный пример откроет файл и установит канал к нему при вызове метода `Files.newByteChannel()`. Как показано в предыдущем разделе, общая форма метода `newByteChannel()` такова.

`static SeekableByteChannel newByteChannel(Path` *`путь`*`, OpenOption ... `*`как`*`) throws IOException`

Метод возвращает объект интерфейса `SeekableByteChannel`, инкапсулирующий канал для работы с файлом. Чтобы открыть файл для вывода, параметр *`как`* должен передать значение `StandardOpenOption.WRITE`. Если файл еще не существует и его необходимо создать, то следует определить также значение `StandardOpenOption.CREATE`. (Другие доступные значения параметра перечислены в табл. 20.7.) Как описано в предыдущем разделе, интерфейс `SeekableByteChannel` описывает канал, применяемый для файловых операций. Его реализует класс `FileChannel`. Когда используется файловая система по умолчанию, возвращаемый объект может быть приведен к типу `FileChannel`. Завершив работу с каналом, его следует закрыть.

Один из способов записи в файл через канал подразумевает использование явных вызовов метода `write()`. Сначала получите объект интерфейса `Path` для файла, а затем откройте его вызовом метода `newByteChannel()`, приводя ре-

зультат к типу FileChannel. Затем зарезервируйте байтовый буфер и запишите в него данные. Прежде чем данные будут записаны в файл, для буфера следует вызвать метод rewind(), чтобы обнулить его текущую позицию. (Каждая операция вывода в буфер увеличивает его текущую позицию. Поэтому перед записью в файл ее следует возвратить в исходное состояние.) Далее вызовите для канала метод write(), передав ему буфер. Эту процедуру демонстрирует следующая программа. Она записывает алфавит в файл по имени test.txt.

```
// Запись в файл с использованием NIO. Требует JDK 7.

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 для файла в пределах блока try-с-ресурсами.
        try ( FileChannel fChan = (FileChannel)
                Files.newByteChannel(Paths.get("test.txt"),
                                     StandardOpenOption.WRITE,
                                     StandardOpenOption.CREATE) )
        {
            // Создать буфер.
            ByteBuffer mBuf = ByteBuffer.allocate(26);

            // Записать несколько байтов в буфер.
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));

            // Подготовить буфер для записи.
            mBuf.rewind();

            // Запись буфера в выходной файл.
            fChan.write(mBuf);

        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch (IOException e) {
            System.out.println("I/O Error: " + e);
            System.exit(1);
        }
    }
}
```

Имеет смысл подчеркнуть важный аспект этой программы. Как упоминалось, после записи данных в объект байтового буфера mBuf, однако прежде, чем они будут записаны в файл, для объекта mBuf осуществляется вызов метода rewind(). Это необходимо для обнуления текущей позиции после записи данных в буфер mBuf. Помните: каждый вызов метода put() объекта mBuf передвигает текущую позицию. Поэтому текущую позицию необходимо вернуть в начало буфера, прежде чем вызывать метод write(). Если это не будет сделано, то метод write() будет считать, что никаких данных в буфере нет.

Еще один способ обнуления буфера между операциями ввода и вывода подразумевает вызов метода flip() вместо метода rewind(). Метод flip() устанавливает для текущей позиции нулевое значение, а для предела — значение предыдущей текущей позиции. В приведенном выше примере, поскольку емкость буфера

совпадала с его пределом, метод `flip()` можно использовать вместо метода `rewind()`. Но эти два метода не являются взаимозаменяемыми во всех случаях.

Как правило, следует обнулять буфер между любыми операциями чтения и записи. Например, с учетом приведенного выше примера, следующий цикл запишет алфавит в файл три раза. Обратите особое внимание на вызовы метода `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));

    // Подготовить буфер для записи.
    mBuf.rewind();

    // Запись буфера в выходной файл.
    fChan.write(mBuf);

    // Подготовить буфер для записи снова.
    mBuf.rewind();
}
```

Обратите внимание: метод `rewind()` вызывается между каждой операцией чтения и записи.

Кроме того, когда буфер будет записываться в файл, первые 26 байт в файле будут содержать вывод. Если файл `test.txt` существовал ранее, то после выполнения программы первые 26 байт файла `test.txt` будут содержать алфавит, а остальная часть файла останется неизменной.

Еще один способ записи в файл подразумевает его сопоставление с буфером. Преимущество этого подхода заключается в том, что занесенные в буфер данные будут автоматически записаны в файл. Никаких явных операций записи не нужно. Чтобы сопоставить и записать содержимое файла, мы будем использовать такую общую процедуру. Сначала получите объект интерфейса `Path`, который инкапсулирует файл, а затем создайте канал к этому файлу, вызвав метод `Files.newByteChannel()` и передав ему объект интерфейса `Path`. Приведите ссылку, возвращенную методом `newByteChannel()`, к типу `FileChannel`. Затем сопоставьте канал с буфером при вызове метода `map()` для канала. Метод `map()` был подробно описан в предыдущем разделе, а здесь он упомянут для вашего удобства. Вот его общая форма.

```
MappedByteBuffer map(FileChannel.MapMode как, long позиция, long размер)
throws IOException
```

Метод `map()` сопоставляет данные в файле с буфером в памяти. Значение параметра *как* определяет разрешенные операции. Чтобы писать в файл, параметр *как* должен содержать значение `MapMode.READ_WRITE`. Положение начала сопоставления в пределах файла определяет параметр *позиция*, а количество сопоставляемых байтов — параметр *размер*. Возвращается ссылка на этот буфер. Как только файл сопоставлен с буфером, вы можете писать в буфер данные и они автоматически будут записываться в файл. Поэтому никаких явных операций записи в канал не нужно.

Вот предыдущая программа, переделанная так, чтобы использовался сопоставленный файл. Обратите внимание на то, что в вызове метода `newByteChannel()` в параметр добавлено значение `StandardOpenOption.READ`. Дело в том, что сопоставленный буфер может использоваться или только для чтения, или для чтения и записи. Таким образом, для записи в сопоставленный буфер канал должен быть открыт и для чтения, и для записи.

```
// Запись в сопоставленный файл. Требует JDK 7.

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 для файла в пределах блока try-с-ресурсами.
        try ( FileChannel fChan = (FileChannel)
              Files.newByteChannel(Paths.get("test.txt"),
                                   StandardOpenOption.WRITE,
                                   StandardOpenOption.READ,
                                   StandardOpenOption.CREATE) )
        {

            // Затем 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("Path Error " + e);
        } catch (IOException e) {
            System.out.println("I/O Error " + e);
        }
    }
}
```

Как можно заметить, здесь нет никаких явных операций записи непосредственно в канал. Поскольку буфер `mBuf` сопоставлен с файлом, изменения в буфере автоматически отражаются в основном файле.

Копирование файла с использованием NIO

Система NIO упрощает несколько типов файловых операций. Хотя мы не можем рассмотреть их все, пример даст вам общее представление о том, что доступно. Следующая программа копирует файл непосредственно одним методом NIO `copy()`, являющимся статическим методом класса `Files`. Он имеет несколько форм. Вот та, которую мы будем использовать.

```
static Path copy(Path ист, Path назн, CopyOption ... как) throws
IOException
```

Файл, определенный параметром *ист*, копируется в файл, определенный параметром *назн*. То, как будет выполняться копирование, определяет параметр *как*. Поскольку это параметр для аргумента переменной длины, он может отсутствовать. Если он определен, то может передать одно или несколько следующих значений, которые допустимы для всех файловых систем.

`StandardCopyOption.COPY_ATTRIBUTES`	Требовать копирования атрибутов файла
`StandardLinkOption.NOFOLLOW_LINKS`	Не следовать по символическим ссылкам
`StandardCopyOption.REPLACE_EXISTING`	Перезаписать прежний файл

В зависимости от реализации, могут поддерживаться и другие возможности.

Следующая программа демонстрирует метод `copy()`. Исходный и результирующие файлы определяются в командной строке, причем исходный файл указывается первым. Обратите внимание на размер программы. Вы могли бы сравнить эту версию программы копирования файла с аналогом из главы 13. В результате ока-

жется, что та часть программы, которая фактически копирует файл, существенно короче в версии NIO, представленной здесь.

```
// Копирование файла с использованием NIO. Требует JDK 7.
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("Usage: Copy from to");
            return;
        }

        try {
            Path source = Paths.get(args[0]);
            Path target = Paths.get(args[1]);

            // Скопировать файл.
            Files.copy(source, target,
                       StandardCopyOption.REPLACE_EXISTING);

        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch (IOException e) {
            System.out.println("I/O Error " + e);
        }
    }
}
```

Использование системы NIO для потокового ввода-вывода

Начиная с JDK 7 вы можете использовать систему NIO для открытия потока ввода-вывода. Получив объект интерфейса `Path`, откройте файл, вызвав статический метод `newInputStream()` или `newOutputStream()`, определенные в классе `Files`. Эти методы возвращают поток, связанный с определенным файлом. В любом случае поток затем может быть использован так, как описано в главе 19; применимы те же методики. Преимущество использования объекта интерфейса `Path` для открытия файла в том, что доступны все средства системы NIO.

Чтобы открыть файл для потокового ввода, используйте метод `Files.newInputStream()`. Он имеет следующую форму.

```
static InputStream newInputStream(Path путь, OpenOption ... как) throws
IOException
```

Здесь параметр *путь* определяет открываемый файл, а параметр *как* — то, как файл будет открыт. Это должно быть одно или несколько значений, определенных описанным ранее классом `StandardOpenOption`. (Конечно, применимы только те значения, которые относятся к потоку ввода.) Если параметр не определен, то файл открывается так, как будто было передано значение `StandardOpenOption.READ`.

После открытия вы можете использовать любой из методов, определенных в классе `InputStream`. Например, вы можете использовать метод `read()` для чтения байтов из файла.

Следующая программа демонстрирует использование потокового ввода-вывода на базе NIO. Это программа `ShowFile` из главы 13, переделанная так, чтобы для открытия файла и получения потока использовались средства NIO. Как можно заметить, это очень похоже на первоначальный вариант, за исключением использования интерфейса `Path` и метода `newInputStream()`.

```
/* Отображает текстовый файл, используя потоковый код NIO.
   Требует JDK 7.

   Чтобы использовать эту программу, укажите имя файла, который
   хотите просмотреть. Например, чтобы просмотреть файл по имени
   TEST.TXT, используйте следующую командную строку.

   java ShowFile TEST.TXT
*/

import java.io.*;
import java.nio.file.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;

        // Сначала удостовериться, что имя файла было указано.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile filename");
            return;
        }

        // Открыть файл и получить связанный с ним поток.
        try (InputStream fin = Files.newInputStream(Paths.get(args[0])))
        {
            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch(IOException e) {
            System.out.println("I/O Error " + e);
        }
    }
}
```

Поскольку поток, возвращенный методом `newInputStream()`, является обычным, он применяется как любой другой поток. Например, вы можете заключить поток внутрь буферизованного потока, такого как поток класса `BufferedInputStream`, чтобы обеспечить буферизацию так, как показано далее.

```
new BufferedInputStream(Files.newInputStream(Paths.get(args[0])))
```

Теперь все операции чтения будут автоматически буферизованы.

Чтобы открыть файл для вывода, используйте метод `Files.newOutputStream()`. Он имеет следующую форму.

```
static OutputStream newOutputStream(Path путь, OpenOption ... как)
throws IOException
```

Здесь параметр *путь* определяет открываемый файл, а параметр *как* — то, как файл будет открыт. Это должно быть одно или несколько значений, определенных

описанным ранее классом StandardOpenOption. (Конечно, применимы только те значения, которые относятся к потоку вывода.) Если параметр не определен, то файл открывается так, как будто были переданы значения StandardOpenOption.WRITE, StandardOpenOption.CREATE и StandardOpenOption.TRUNCATE_EXISTING.

Способ использования метода newOutputStream() подобен описанному ранее для метода newInputStream(). После открытия можете использовать любой метод, определенный в классе OutputStream. Например, можете использовать метод write() для записи байтов в файл. Вы можете также заключить поток в байтовый поток класса BufferedOutputStream, чтобы буферизовать его.

Следующая программа демонстрирует метод newOutputStream() в действии. Она записывает алфавит в файл по имени test.txt. Обратите внимание на использование буферизованного ввода-вывода.

```
// Демонстрация потокового вывода на базе NIO. Требует JDK 7.

import java.io.*;
import java.nio.file.*;

class NIOStreamWrite {
    public static void main(String args[])
    {
        // Открыть файл и получить связанный с ним поток.
        try (OutputStream fout =
                  new BufferedOutputStream(
                      Files.newOutputStream(Paths.get("test.txt"))) )
        {
            // Записать в поток несколько байтов.
            for(int i=0; i < 26; i++)
                fout.write((byte)('A' + i));

        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}
```

Использование системы NIO для операций файловой системы

В начале главы 19 был исследован класс File, расположенный в пакете java.io. Как упоминалось, класс File имеет дело с файловой системой и различными атрибутами файла, такими как только для чтения, скрытый и т.д. Он также используется для получения информации о пути файла. С появлением комплекта JDK 7 интерфейсы и классы, определенные системой NIO.2, предлагают лучший способ выполнения этих операций. К их преимуществам относятся поддержка символических ссылок, лучшая поддержка обхода дерева каталогов, улучшенная обработка метаданных и многое другое. В следующих разделах приведены примеры двух популярных операций файловой системы: получение информации о пути и файле, а также о содержимом каталога.

Помните! Если хотите заменить устаревший код, использующий класс java.io.File, на новый, использующий интерфейс Path, можете использовать метод toPath(), чтобы получить экземпляр интерфейса Path из экземпляра класса File. Метод toPath() был добавлен к классу File в JDK 7.

Получение информации о пути и файле

Информация о пути может быть получена при помощи методов, определенных интерфейсом `Path`. Некоторые, связанные с файлом атрибуты, описанные интерфейсом `Path` (такие, как скрытый), получают при помощи методов, определенных в классе `Files`. Здесь используются такие методы интерфейса `Path`, как `getName()`, `getParent()` и `toAbsolutePath()`, а класс `Files` предоставляет такие методы, как `isExecutable()`, `isHidden()`, `isReadable()`, `isWritable()` и `exists()`. Они представлены в приведенных выше табл. 20.5 и 20.6.

Внимание! Такие методы, как `isExecutable()`, `isReadable()`, `isWritable()` и `exists()`, следует использовать осторожно, поскольку состояние файловой системы после вызова может измениться, в программе может произойти сбой. В такой ситуации может иметь значение защита.

Другие атрибуты файла получают, затребовав их список, создаваемый при вызове метода `Files.readAttributes()`. В программе этот метод применяется для получения связанного с файлом объекта интерфейса `BasicFileAttributes`, но общий подход применим и к другим типам атрибутов.

Следующая программа демонстрирует некоторые методы интерфейса `Path` и класса `Files` наряду с несколькими методами, предоставленными интерфейсом `BasicFileAttributes`. Эта программа подразумевает, что файл по имени `test.txt` существует в каталоге `examples`, внутри текущего каталога.

```
// Получить информацию о пути и файле.
// Требует JDK 7.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class PathDemo {
    public static void main(String args[]) {
        Path filepath = Paths.get("examples\\test.txt");

        System.out.println("File Name: " + filepath.getName(1));
        System.out.println("Path: " + filepath);
        System.out.println("Absolute Path: " +
                           filepath.toAbsolutePath());
        System.out.println("Parent: " + filepath.getParent());

        if(Files.exists(filepath))
            System.out.println("File exists");
        else
            System.out.println("File does not exist");

        try {
            if(Files.isHidden(filepath))
                System.out.println("File is hidden");
            else
                System.out.println("File is not hidden");
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
        }

        Files.isWritable(filepath);
        System.out.println("File is writable");

        Files.isReadable(filepath);
        System.out.println("File is readable");
```

```
        try {
            BasicFileAttributes attribs =
               Files.readAttributes(filepath, BasicFileAttributes.class);

            if(attribs.isDirectory())
                System.out.println("The file is a directory");
            else
                System.out.println("The file is not a directory");

            if(attribs.isRegularFile())
                System.out.println("The file is a normal file");
            else
                System.out.println("The file is not a normal file");

            if(attribs.isSymbolicLink())
                System.out.println("The file is a symbolic link");
            else
                System.out.println("The file is not a symbolic link");

            System.out.println("File last modified: " +
                               attribs.lastModifiedTime());
            System.out.println("File size: " + attribs.size() +
                               " Bytes");
        } catch(IOException e) {
            System.out.println("Error reading attributes: " + e);
        }
    }
}
```

Если запустить эту программу из каталога `MyDir`, внутри которого есть каталог `examples`, содержащий файл `test.txt`, то увидите вывод, подобный представленному ниже. (Конечно, размер файла и время будут иными.)

```
File Name: test.txt
Path: examples\test.txt
Absolute Path: C:\MyDir\examples\test.txt
Parent: examples
File exists
File is not hidden
File is writable
File is readable
The file is not a directory
The file is a normal file
The file is not a symbolic link
File last modified: 2010-09-01T18:20:46.380445Z
File size: 18 Bytes
```

Если вы используете компьютер с файловой системой FAT (т.е. файловой системой DOS), то могли бы попробовать использовать методы, определенные интерфейсом `DosFileAttributes`. Если вы используете систему, совместимую с POSIX, то попробуйте использовать методы, определенные интерфейсом `PosixFileAttributes`.

Перечисление содержимого каталога

Если путь описывает каталог, то вы можете прочитать содержимое этого каталога, используя статические методы, определенные в классе `Files`. Для этого вы получаете сначала поток каталога, вызвав метод `newDirectoryStream()` при передаче ему объекта интерфейса `Path`, описывающего каталог. Одна из форм метода `newDirectoryStream()` приведена ниже.

```
static DirectoryStream<Path> newDirectoryStream(Path путьКкаталогу)
throws IOException
```

Здесь аргумент *путьКкаталогу* инкапсулирует путь к каталогу. Метод возвращает объект `DirectoryStream<Path>`, применяемый для получения содержимого каталога. Он передает исключение `IOException`, в случае ошибки ввода-вывода, и исключение `NotDirectoryException` (класс которого происходит от класса `IOException`), если определенный путь не является каталогом. Возможна также передача исключения `SecurityException`, если доступ к каталогу не разрешен.

Поскольку объект `DirectoryStream<Path>` реализует интерфейс `AutoCloseable`, он может контролироваться оператором `try` *с ресурсами*. Он также реализует интерфейс `Iterable<Path>`. Это значит, что вы можете получить содержимое каталога, перебрав объект `DirectoryStream`. При переборе каждая запись каталога представляется экземпляром интерфейса `Path`. Простейший способ перебора объекта `DirectoryStream` — использование цикла `for` в стиле for-each. Однако следует уяснить, что итератор, реализованный объектом `DirectoryStream<Path>`, может быть получен только однажды для каждого экземпляра. Таким образом, метод `iterator()` может быть вызван только однажды и цикл `for` в стиле for-each может быть выполнен только один раз.

Следующая программа отображает содержимое каталога по имени `MyDir`.

```
// Отображает каталог. Требует JDK 7.

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 обработать поток каталога в пределах блока try.
        try ( DirectoryStream<Path> dirstrm =
              Files.newDirectoryStream(Paths.get(dirname)) )
        {
            System.out.println("Directory of " + dirname);

            // Поскольку DirectoryStream реализует интерфейс Iterable,
            // мы можем использовать цикл "foreach" для отображения
            // каталога.
            for(Path entry : dirstrm) {
                BasicFileAttributes attribs =
                  Files.readAttributes(entry, BasicFileAttributes.class);

                if(attribs.isDirectory())
                    System.out.print("<DIR> ");
                else
                    System.out.print("      ");

                System.out.println(entry.getName(1));
            }
        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch(NotDirectoryException e) {
            System.out.println(dirname + " is not a directory.");
        } catch (IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}
```

Вот пример вывода этой программы.

```
Directory of \MyDir
      DirList.class
      DirList.java
<DIR> examples
      Test.txt
```

Вы можете отфильтровать содержимое каталога двумя способами. Самый простой – использование такой версии метода `newDirectoryStream()`.

```
static DirectoryStream<Path> newDirectoryStream(Path путьКкаталогу,
String шаблон) throws IOException
```

В этой версии будут получены только те файлы, имена которых соответствуют шаблону, определенному параметром *шаблон*. Для параметра *шаблон* вы можете определить или полное имя файла, или объект `glob`. Объект `glob` – это строка, определяющая общий шаблон, которому будет соответствовать один или несколько файлов, и содержащая знакомые символы * и ?. Они соответствуют любому количеству любых символов и любому символу соответственно. Ниже приведены соответствия в пределах объекта `glob`.

**	Соответствует любому количеству различных символов в каталоге
[*символы*]	Соответствует любому символу в *символы*. Символы * и ? в пределах *символы* будут рассматриваться как обычные символы, а не как символы шаблона. При помощи дефиса может быть указан диапазон, такой как [x-z]
{*списокglob*}	Соответствует любому объекту `glob`, заданному в разделяемом запятыми списке объектов `glob` в *списокglob*

Вы можете определить символы * и ?, используя последовательности * и \?. Чтобы определить символ \, используйте последовательность \\. Вы можете поэкспериментировать с объектом `glob`, подставляя его в вызов метода `newDirectoryStream()` предыдущей программы.

```
Files.newDirectoryStream(Paths.get(dirname), "{Path,Dir}*.{java,class}")
```

В результате получится поток каталога, содержащий только те файлы, имена которых начинаются или с `"Path"` или `"Dir"` и имеют расширение `"java"` или `"class"`. Таким образом, это соответствовало бы таким именам, как `DirList.java` и `PathDemo.java`, но не `MyPathDemo.java`.

Еще один способ фильтрации каталога заключается в использовании такой версии метода `newDirectoryStream()`.

```
static DirectoryStream<Path> newDirectoryStream(Path путьКкаталогу,
DirectoryStream.Filter<? super Path> файловыйфильтр) throws IOException
```

Здесь `DirectoryStream.Filter` – это интерфейс, определяющий следующий метод.

```
boolean accept(T элемент) throws IOException
```

В данном случае типом `T` будет `Path`. Если вы хотите включить *элемент* в список, возвращается значение `true`. В противном случае возвращается значение `false`. Эта форма метода `newDirectoryStream()` предоставляет возможность фильтрации каталога на основании чего-то отличного от имени файла. Например, вы можете фильтровать на основании размера, даты создания, даты модификации или атрибута.

Следующая программа демонстрирует этот процесс. Здесь перечислены только те файлы, которые допускают запись.

```
// Отображает только те файлы 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, возвращающий true только для
        // записываемых файлов.
        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 использовать поток каталога для
        // записываемых файлов.
        try (DirectoryStream<Path> dirstrm =
            Files.newDirectoryStream(Paths.get(dirname), how) )
        {
            System.out.println("Directory of " + dirname);

            for(Path entry : dirstrm) {
                BasicFileAttributes attribs =
                  Files.readAttributes(entry, BasicFileAttributes.class);

                if(attribs.isDirectory())
                    System.out.print("<DIR> ");
                else
                    System.out.print("      ");

                System.out.println(entry.getName(1));
            }
        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch(NotDirectoryException e) {
            System.out.println(dirname + " is not a directory.");
        } catch (IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}
```

Использование метода `walkFileTree()` для перечисления дерева каталогов

В предыдущих примерах мы получали содержимое только одного каталога. Но иногда необходимо получить список файлов в дереве каталогов. В прошлом это было настоящей проблемой, но система NIO.2 существенно облегчает это, поскольку теперь вы можете использовать определенный в классе `Files` метод `walkFileTree()`, способный обработать дерево каталогов. Этот метод имеет две формы; в данной главе используется такая.

```
static Path walkFileTree(Path корень, FileVisitor<? extends Path> fv)
throws IOException
```

Начальный пункт обхода каталога передается в параметре *корень*. Экземпляр интерфейса `FileVisitor` передается в параметре *fv*. Реализация интерфейса `FileVisitor` определяет способ обхода дерева каталогов, а также позволяет обращаться к информации каталога. При ошибке ввода-вывода передается исключение `IOException`. Возможна также передача исключения `SecurityException`.

Интерфейс `FileVisitor` определяет то, как посещаются файлы при обходе дерева каталогов. Это обобщенный интерфейс, который объявляется так.

```
interface FileVisitor<T>
```

Для использования в методе `walkFileTree()`, параметр типа *T* будет иметь тип интерфейса `Path` (или любой тип, производный от него). В интерфейсе `FileVisitor` определены методы, перечисленные в табл. 20.11.

Таблица 20.11. Методы, определенные в интерфейсе `FileVisitor`

Метод	Описание
`FileVisitResult postVisitDirectory(T` *каталог*`, IOException` *исключение*`) throws IOException`	Вызывается после посещения каталога. Каталог передается в параметре *каталог*, а любое исключение `IOException` — в параметре *исключение*. Если параметр *исключение* содержит значение `null`, значит, никакого исключения не произошло. Результат возвращается
`FileVisitResult preVisitDirectory(T` *каталог*`, BasicFileAttributes` *атрибуты*`) throws IOException`	Вызывается перед посещением каталога. Каталог передается в параметре *каталог*, а связанные с ним атрибуты — в параметре *атрибуты*. Результат возвращается. Чтобы исследовать каталог, возвратите значение `FileVisitResult.CONTINUE`
`FileVisitResult visitFile(T` *файл*`, BasicFileAttributes` *атрибуты*`) throws IOException`	Вызывается при посещении файла. Файл передается в параметре *файл*, а связанные с ним атрибуты — в параметре *атрибуты*. Результат возвращается
`FileVisitResult visitFileFailed(T` *файл*`, IOException` *исключение*`) throws IOException`	Вызывается при неудачной попытке посещения файла. Файл, который потерпел неудачу, передается в параметре *файл*, а исключение `IOException` — в параметре *исключение*. Результат возвращается

Обратите внимание на то, что каждый метод возвращает значение перечисления `FileVisitResult`. Это перечисление определяет следующие значения.

`CONTINUE`	`SKIP_SIBLINGS`	`SKIP_SUBTREE`	`TERMINATE`

Вообще, для продолжения обхода каталога и каталогов внутри него метод должен возвратить значение `CONTINUE`. Для метода `preVisitDirectory()` возвратите значение `SKIP_SIBLINGS`, чтобы пропустить каталог и его содержимое, а также предотвратить вызов метода `postVisitDirectory()`. Чтобы пропустить только каталог и подкаталоги, возвратите значение `SKIP_SUBTREE`. Чтобы остановить обход каталога, возвратите значение `TERMINATE`.

Конечно, вполне можно создать собственный класс посещения, который реализует эти методы, определенные интерфейсом `FileVisitor`, но обычно вы не будете поступать так, поскольку предоставляется его простая реализация, класс `SimpleFileVisitor`. Достаточно переопределить заданную по умолчанию реализацию метода или методов, которые вас интересуют. Вот краткий пример, который иллюстрирует этот процесс. Он отображает все файлы в дереве каталогов, корнем которого является каталог `\MyDir`. Обратите внимание на размер этой программы.

```
// Простой пример использования метода walkFileTree() для отображения
// дерева каталогов. Требует JDK 7.
```

```
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

// Создать специальную версию SimpleFileVisitor, переопределяющую
// 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("Directory tree starting with " +
                           dirname + ":\n");
        try {
            Files.walkFileTree(Paths.get(dirname), new MyFileVisitor());
        } catch (IOException exc) {
            System.out.println("I/O Error");
        }
    }
}
```

Вот пример вывода программы, выполненной для того же каталога `MyDir`, что и ранее. В этом примере вложенный каталог `examples` содержит только один файл по имени `MyProgram.java`.

```
Directory tree starting with \MyDir:

\MyDir\DirList.class
\MyDir\DirList.java
\MyDir\examples\MyProgram.java
\MyDir\Test.txt
```

В этой программе класс `MyFileVisitor` расширяет класс `SimpleFileVisitor`, переопределяя только метод `visitFile()`. В этом примере метод `visitFile()` просто отображает их, но вполне можно достичь и более сложных функциональных возможностей. Например, вы могли бы фильтровать файлы или выполнять с ними такие действия, как их копирование на устройство резервирования. Для простоты использовался именованный класс, переопределяющий метод `visitFile()`, но вполне можно также использовать анонимный внутренний класс.

И последний момент: используя `java.nio.file.WatchService`, можно отследить изменения в каталоге.

Примеры использования каналов до JDK 7

Прежде чем завершить эту главу, следует рассмотреть еще один аспект системы NIO. В приведенных выше разделах использовались некоторые новые средства, добавленные в систему NIO с появлением JDK 7. Однако осталось еще много кода, написанного до JDK 7, который необходимо поддерживать, а возможно, и преобразовывать для использования новых средств. Поэтому следующие разделы де-

монстрируют чтение и запись файлов с использованием системы NIO до JDK 7. Некоторые из приведенных выше примеров переделаны так, чтобы использовать предыдущие средства NIO, а не новые (NIO.2). Это значит, что примеры в этом разделе будут работать с версиями Java до JDK 7.

Основное отличие между прежним кодом NIO и новым заключается в интерфейсе `Path`, который был добавлен с появлением JDK 7. Таким образом, прежний код не использует интерфейс `Path` для описания файла или открытия канала к нему. Кроме того, прежний код не использует операторы *try-с-ресурсами*, поскольку автоматическое управление ресурсами также было добавлено только в JDK 7.

Помните! В примерах этого раздела описывается работа устаревшего кода NIO. Материал этого раздела предназначен для тех программистов, которые продолжают работать с кодом, написанным до появления JDK 7, или используют прежний компилятор. Новый код должен использовать средства NIO, добавленные JDK 7.

Чтение из файла до JDK 7

Здесь приведено два предыдущих примера канального ввода из файла, переделанных для использования средств только до JDK 7. В первом примере выполняется чтение файла при резервировании буфера вручную и последующем явном выполнении операций чтения. Во втором примере осуществляется сопоставление файла, автоматизирующее процесс.

При использовании версий Java до JDK 7, для чтения файла с использованием канала и зарезервированного вручную буфера, вы сначала открываете файл для ввода, используя объект класса `FileInputStream`, точно так же как описано в главе 19. Затем получаете канал к этому файлу, вызвав метод `getChannel()` для объекта класса `FileInputStream`. Его общая форма такова.

```
FileChannel getChannel()
```

Она возвращает объект класса `FileChannel`, инкапсулирующий канал для файловых операций. Затем вызов метода `allocate()` резервирует буфер. Поскольку файловые каналы работают с байтовыми буферами, вы будете использовать метод `allocate()`, определенный в классе `ByteBuffer`, как было описано ранее.

Следующая программа демонстрирует чтение и отображение файла по имени `test.txt` через канал с использованием явных операций ввода для версий Java до JDK 7.

```
// Использование каналов для чтения файла. Версия до JDK 7.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        FileInputStream fIn = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;
        int count;

        try {
            // Сначала открыть файл для ввода.
            fIn = new FileInputStream("test.txt");

            // Затем получить канал к этому файлу.
            fChan = fIn.getChannel();

            // Зарезервировать буфер.
```

```
            mBuf = ByteBuffer.allocate(128);

            do {
            // Читать в буфер.
            count = fChan.read(mBuf);

            // Остановиться при достижении конца файла.
            if(count != -1) {

                // Подготовить буфер для чтения.
                mBuf.rewind();

                // Читать байты в буфер и отображать 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("I/O Error " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch(IOException e) {
                System.out.println("Error Closing Channel.");
            }
            try {
                if(fIn != null) fIn.close(); // закрыть файл
            } catch(IOException e) {
                System.out.println("Error Closing File.");
            }
        }
    }
}
```

Обратите внимание на то, что в этой программе файл открывается с использованием конструктора класса `FileInputStream`, а ссылка на этот объект присваивается объекту `fIn`. Затем, при вызове метода `getChannel()` объекта `fIn`, создается канал соединения с файлом. После этого программа работает, как в версиях для JDK 7, представленных ранее. Короче говоря, затем программа вызывает метод `allocate()` класса `ByteBuffer`, чтобы зарезервировать буфер для размещения содержимого файла при чтении. Используется байтовый буфер, поскольку класс `FileChannel` работает с байтами. Ссылка на этот буфер хранится в объекте `mBuf`. Затем содержимое файла читается в буфер `mBuf` по одному буферу за раз при помощи метода `read()`. Количество прочитанных байтов хранится в переменной `count`. Затем буфер возобновляется вызовом метода `rewind()`. Этот вызов необходим, поскольку после вызова метода `read()` текущая позиция буфера находится в конце, а ее следует вернуть в начало буфера, чтобы байты из буфера `mBuf` могли быть прочитаны при вызове метода `get()`. Когда будет достигнут конец файла, метод `read()` возвратит значение `-1`. Когда это произойдет, программа завершит работу явно, закрыв канал и файл.

Другой способ чтения файла состоит в его сопоставлении с буфером. Как объяснялось ранее, основное преимущество этого подхода в том, что буфер автоматически получает содержимое файла. Никаких явных операций чтения не нужно. Чтобы сопоставить и прочитать содержимое файла, используя средства NIO до JDK 7, откройте сначала файл при помощи класса `FileInputStream`. Затем получите канал к этому файлу при вызове метода `getChannel()` для объекта файла.

После этого сопоставьте канал с буфером при вызове метода `map()` объекта класса `FileChannel`. Метод `map()` работает, как описано ранее.

Следующая программа — это предыдущий пример, переделанный так, чтобы для сопоставления файла использовались только средства до JDK 7.

```
// Использование сопоставления для чтения файла. Версия до JDK 7.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelRead {
    public static void main(String args[]) {
        FileInputStream fIn = null;
        FileChannel fChan = null;
        long fSize;
        MappedByteBuffer mBuf;

        try {
            // Сначала открыть файл для ввода.
            fIn = new FileInputStream("test.txt");

            // Затем получить канал к этому файлу.
            fChan = fIn.getChannel();

            // Получить размер файла.
            fSize = fChan.size();

            // Теперь сопоставить файл с буфером.
            mBuf = fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

            // Читать и отображать байты из буфера.
            for(int i=0; i < fSize; i++)
                System.out.print((char)mBuf.get());
        } catch (IOException e) {
            System.out.println("I/O Error " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch(IOException e) {
                System.out.println("Error Closing Channel.");
            }
            try {
                if(fIn != null) fIn.close(); // закрыть файл
            } catch(IOException e) {
                System.out.println("Error Closing File.");
            }
        }
    }
}
```

В программе файл открывается при помощи конструктора класса `FileInputStream`, а ссылка на этот объект присваивается объекту `fIn`. Канал, подключенный к файлу, создается при вызове метода `getChannel()` объекта `fIn`. Затем выясняется размер файла. Далее, при вызове метода `map()`, весь файл сопоставляется с областью в памяти, а ссылка на этот буфер сохраняется в объекте `mBuf`. Байты из буфера `mBuf` читаются при вызове метода `get()`.

Запись в файл до JDK 7

В этом разделе два предыдущих примера канального вывода в файл переделаны так, чтобы использовались только средства до JDK 7. В первом примере выполня-

ется запись в файл при резервировании буфера вручную и последующем явном выполнении операций записи. Во втором примере используется сопоставление файла, автоматизирующее процесс. В обоих случаях ни интерфейс `Path`, ни оператор `try-с-ресурсами` не применяются, поскольку их не было в Java до JDK 7.

При использовании версий Java до JDK 7, для записи в файл применяется канал и зарезервированный вручную буфер при предварительном открытии файла для вывода. Для этого создается объект класса `FileOutputStream`, как описано в главе 19. Затем вы получите канал к файлу при вызове метода `getChannel()` и зарезервируете байтовый буфер при вызове метода `allocate()`, как описано в предыдущем разделе. Далее поместите данные, которые хотите записать, в этот буфер и вызовите метод `write()` канала. Следующая программа демонстрирует эту процедуру. Она записывает алфавит в файл по имени `test.txt`.

```
// Запись в файл с использованием NIO. Версия до JDK 7.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelWrite {
    public static void main(String args[]) {
        FileOutputStream fOut = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;

        try {
            // Сначала открыть файл для вывода.
            fOut = new FileOutputStream("test.txt");

            // Затем получить канал к файлу для вывода.
            fChan = fOut.getChannel();

            // Создать буфер.
            mBuf = ByteBuffer.allocate(26);

            // Записать несколько байтов в буфер.
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));

            // Подготовить буфер для записи.
            mBuf.rewind();

            // Запись буфера в выходной файл.
            fChan.write(mBuf);

        } catch (IOException e) {
            System.out.println("I/O Error " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch(IOException e) {
                System.out.println("Error Closing Channel.");
            }
            try {
                if(fOut != null) fOut.close(); // закрыть файл
            } catch(IOException e) {
                System.out.println("Error Closing File.");
            }
        }
    }
}
```

Вызов метода rewind() объекта mBuf необходим для возвращения текущей позиции буфера mBuf в нуль после записи данных. Помните, что каждый вызов метода put() перемещает текущую позицию. Поэтому прежде, чем вызвать метод write(), необходимо установить текущую позицию в начало буфера. В противном случае метод write() посчитает, что никаких данных в буфере нет.

При использовании версий Java до JDK 7, для записи в файл выполняется сопоставление файла с буфером следующим образом. Сначала откройте файл для операций чтения и записи при создании объекта класса RandomAccessFile. Для файла необходимо разрешение на чтение и запись. Затем сопоставьте этот файл с буфером при вызове метода map() данного объекта. Далее запишите данные в буфер. Поскольку буфер сопоставлен с файлом, любые изменения в этом буфере автоматически отражаются в файле. Таким образом, никаких явных операций записи в канал не нужно.

Вот приведенная выше программа, переделанная для использования сопоставления файла.

```
// Запись в сопоставленный файл. Версия до JDK 7.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelWrite {
    public static void main(String args[]) {
        RandomAccessFile fOut = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;

        try {
            fOut = new RandomAccessFile("test.txt", "rw");

            // Затем получить канал к этому файлу.
            fChan = fOut.getChannel();

            // Затем сопоставить файл с буфером.
            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 (IOException e) {
            System.out.println("I/O Error " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch(IOException e) {
                System.out.println("Error Closing Channel.");
            }
            try {
                if(fOut != null) fOut.close(); // закрыть файл
            } catch(IOException e) {
                System.out.println("Error Closing File.");
            }
        }
    }
}
```

Как можно заметить, здесь нет никаких явных операций записи непосредственно в канал. Поскольку буфер mBuf сопоставляется с файлом, изменения в буфере автоматически отражаются в основном файле.

ГЛАВА 21 Сеть

Как известно читателям, Java – практически синоним программирования для Интернета. Тому есть множество причин, и не последняя из них – способность создавать безопасный, межплатформенный и переносимый код. Однако одна из наиболее важных причин того, что Java является великолепным языком сетевого программирования, кроется в классах, определенных в пакете `java.net`. Они обеспечивают легкие в использовании средства, с помощью которых программисты всех уровней квалификации могут обращаться к сетевым ресурсам.

Эта глава посвящена пакету `java.net`. Важно подчеркнуть, что сети – очень обширная и сложная тема. В настоящей книге невозможно полностью охватить все средства, содержащиеся в пакете `java.net`. Поэтому в данной главе сосредоточим внимание лишь на некоторых основополагающих классах и интерфейсах.

Основы работы с сетью

Прежде чем начать, полезно будет получить представление о ключевых концепциях и терминах, связанных с сетями. В основе сетевой поддержки Java лежит концепция *сокета* (socket); сокет идентифицирует конечную точку сети. Парадигма сокета появилась в версии 4.2 BSD Berkley UNIX в самом начале 80-х гг. По этой причине также используется термин *сокет Беркли*. Сокеты – основа современных сетей, поскольку сокет позволяет отдельному компьютеру обслуживать одновременно как множество разных клиентов, так и множество различных типов информации. Это достигается за счет использования *порта* (port) – нумерованного сокета на определенной машине. Говорят, что серверный процесс "слушает" порт до тех пор, пока клиент не соединится с ним. Сервер в состоянии принять множество клиентов, подключенных к одному и тому же номеру порта, хотя каждый сеанс является уникальным. Чтобы обработать множество клиентских соединений, серверный процесс должен быть многопоточным либо обладать какими-то другими средствами обработки одновременного ввода-вывода.

Сокетные коммуникации происходят по определенному протоколу. *Протокол Интернета* (Internet Protocol – IP) – это низкоуровневый маршрутизирующий протокол, который разбивает данные на небольшие пакеты и посылает их через сеть по определенному адресу, что не гарантирует доставки всех этих пакетов по этому адресу. *Протокол управления передачей* (Transmission Control Protocol – TCP) является протоколом более высокого уровня, обеспечивающим надежную сборку этих пакетов, сортировку и повторную передачу, необходимую для надежной доставки данных. Еще один протокол – *протокол пользовательских дейтаграмм* (User Datagram Protocol – UDP), стоящий непосредственно за протоколом TCP, – может быть использован непосредственно для поддержки быстрой, не требующей постоянного соединения и ненадежной транспортировки пакетов.

Как только соединение установлено, применяется высокоуровневый протокол, зависящий от используемого порта. Протокол TCP/IP резервирует первые 1024 порта для специфических протоколов. Многие из них покажутся вам знакомыми, если вы хоть какое-то время потратили на путешествия по просторам Интернета. Порт номер 21 предназначен для протокола FTP, 23 – для Telnet, 25 – для электронной почты, 43 – для whois, 80 – для протокола HTTP, 119 – для netnews; список можно продолжать. Каждый протокол определяет, как клиент должен взаимодействовать с портом.

Например, HTTP – это протокол, используемый серверами и веб-браузерами для передачи гипертекста и графических изображений. Это довольно простой протокол для базового постраничного просмотра информации, предоставляемой веб-серверами. Посмотрим, как он работает. Когда клиент запрашивает файл с сервера HTTP, это действие известно как *попадание* (hit) и заключается в простой отправке имени файла в определенном формате на предопределенный порт с последующим чтением содержимого этого файла. Сервер также сообщает код состояния, чтобы известить клиента о том, был ли запрос обработан и по какой причине.

Ключевым компонентом Интернета является *адрес*. Каждый компьютер в Интернете обладает собственным адресом. Адрес Интернета представляет собой число, уникально идентифицирующее каждый компьютер в Интернете. Изначально все адреса Интернета состояли из 32-битовых значений, организованных в четыре 8-битовых значения. Адрес такого типа определен IPv4 (Протокол Интернета версии 4). Однако в последнее время на сцену выступает новая схема адресации, называемая IPv6, которая предназначена для того, чтобы поддержать гораздо большее адресное пространство, чем IPv4. К счастью, имея дело с Java, вам обычно не придется беспокоиться о том, используется адрес IPv4 или IPv6, поскольку Java позаботится обо всех деталях.

Точно так же как IP-адрес описывает сетевую иерархию, имя адреса Интернета, называемое *доменным именем*, представляет местонахождение машины в пространстве имен. Например, адрес `www.HerbSchildt.com` относится к верховному домену `com` (зарезервированному для коммерческих сайтов США), имеет имя `HerbSchildt` (по названию компании), а `www` идентифицирует сервер, обрабатывающий веб-запросы. Доменное имя Интернета сопоставляется с IP-адресом при помощи *службы доменных имен* (Domain Name Service – DNS). Это позволяет пользователям работать с доменными именами, в то время как Интернет оперирует IP-адресами.

Сетевые классы и интерфейсы

Язык Java поддерживает протокол TCP/IP как за счет расширения уже имеющихся интерфейсов потокового ввода-вывода, представленных в главе 19, так и за счет добавления средств, необходимых для построения объектов ввода-вывода в сети. Java поддерживает семейства протоколов как TCP, так и UDP. Протокол TCP применяется для надежного потокового ввода-вывода по сети. Протокол UDP поддерживает более простую, а потому быструю модель передачи дейтаграмм от точки к точке. Классы, содержащиеся в пакете `java.net`, перечислены ниже.

`Authenticator`	`Inet6Address`	`ServerSocket`
`CacheRequest`	`InetAddress`	`Socket`
`CacheResponse`	`InetSocketAddress`	`SocketAddress`
`ContentHandler`	`InterfaceAddress`	`SocketImpl`
`CookieHandler`	`JarURLConnection`	`SocketPermission`

CookieManager	MulticastSocket	StandardSocketOption (Добавлено в JDK 7)
DatagramPacket	NetPermission	URI
DatagramSocket	NetworkInterface	URL
DatagramSocketImpl	PasswordAuthentication	URLClassLoader
HttpCookie	Proxy	URLConnection
HttpURLConnection	ProxySelector	URLDecoder
IDN	ResponseCache	URLEncoder
Inet4Address	SecureCacheResponse	URLStreamHandler

Интерфейсы пакета `java.net` перечислены далее.

ContentHandlerFactory	FileNameMap	SocketOptions
CookiePolicy	ProtocolFamily (Добавлено в JDK 7)	URLStreamHandlerFactory
CookieStore	SocketImplFactory	
DatagramSocketImplFactory	SocketOption (Добавлено в JDK 7)	

В следующем разделе рассмотрим основные сетевые классы и продемонстрируем несколько примеров их применения. Как только поймете устройство сетевых классов, сможете строить собственные на их основе.

Класс `InetAddress`

Рассматриваемый класс используется для инкапсуляции как числового IP-адреса, так и его доменного имени. Вы взаимодействуете с классом, используя имя IP-хоста, что намного удобнее и понятнее, чем IP-адрес. Класс `InetAddress` скрывает внутри себя число. Он может работать как с адресами IPv4, так и с IPv6.

Методы-фабрики

Класс `InetAddress` не имеет видимых конструкторов. Чтобы создать объект класса `InetAddress`, следует использовать один из доступных методов-фабрик. *Методы-фабрики* (factory method) — это просто соглашение, в соответствии с которым статические методы класса возвращают экземпляр этого класса. Это используется вместо перегрузки конструктора с различными списками параметров, когда наличие уникальных имен методов делает результат более ясным. Ниже приведено три часто используемых метода-фабрики класса `InetAddress`.

```
static InetAddress getLocalHost( )
    throws UnknownHostException
static InetAddress getByName(String имяХоста)
    throws UnknownHostException
static InetAddress[ ] getAllByName(String имяХоста)
    throws UnknownHostException
```

Метод `getLocalHost()` возвращает объект класса `InetAddress`, представляющий локальный хост, метод `getByName()` — объект класса `InetAddress` хоста, имя которого ему передано. Если эти методы оказываются не в состоянии получить имя хоста, они передают исключение `UnknownHostException`.

Когда одно имя используется для представления нескольких машин в Интернете — это обычное явление. В мире веб-серверов это единственный путь обеспечения не-

которой степени масштабируемости. Метод-фабрика `getAllByName()` возвращает массив класса `InetAddress`, представляющий все адреса, в которые преобразуется конкретное имя. Он также передает исключение `UnknownHostException` в случае, если не может преобразовать имя хотя бы в один адрес.

Класс `InetAddress` также включает метод-фабрику `getDyAddress()`, который принимает IP-адрес и возвращает объект класса `InetAddress`. Причем могут использоваться как адреса IPv4, так и IPv6.

В следующем примере выводятся адреса и имена локальной машины, а также двух сайтов Интернета.

```
// Демонстрация применения 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]);
    }
}
```

Ниже показан вывод, созданный этой программой (конечно, код, который вы увидите на своей машине, может несколько отличаться).

```
default/166.203.115.212
www.HerbSchildt.com/216.92.65.4
www.nba.com/216.66.31.161
www.nba.com/216.66.31.179
```

Методы экземпляра

В классе `InetAddress` имеется также несколько других методов, которые могут быть использованы с объектами, возвращенными методами, о которых мы только что говорили. Некоторые из наиболее часто применяемых методов перечислены в табл. 21.1.

Таблица 21.1. Часто используемые методы класса `InetAddress`

Метод	Описание
`boolean equals(Object` *другое*`)`	Возвращает значение `true`, если объект имеет тот же адрес Интернета, что и *другое*
`byte[] getAddress()`	Возвращает байтовый массив, представляющий IP-адрес в порядке байтов сети
`String getHostAddress()`	Возвращает строку, представляющую адрес хоста, ассоциированного с объектом класса `InetAddress`
`String getHostName()`	Возвращает строку, представляющую имя хоста, ассоциированного с объектом класса `InetAddress`
`boolean isMulticastAddress()`	Возвращает значение `true`, если адрес является групповым, в противном случае возвращает значение `false`
`String toString()`	Возвращает строку, включающую имя хоста и IP-адрес для удобства

Поиск адресов Интернета осуществляется в серии иерархических кешированных служб. Это значит, что ваш локальный компьютер может автоматически соотнести определенное имя с его IP-адресом, как для себя, так и для ближайших серверов. Для всех прочих имен он может обращаться к серверам DNS, откуда получит информацию об IP-адресах. Если такой сервер не имеет информации об определенном адресе, он может обратиться к следующему дистанционному сайту и запросить эту информацию у него. Это может продолжаться вплоть до корневого сервера, и упомянутый процесс может потребовать длительного времени, так что разумно построить структуру вашего кода таким образом, чтобы информация об IP-адресах локально кешировалась и ее не приходилось искать каждый раз заново.

Классы `Inet4Address` и `Inet6Address`

Начиная с версии 1.4 в Java включена поддержка адресов IPv6. В связи с этим были созданы два подкласса класса `InetAddress` — `Inet4Address` и `Inet6Address`. Класс `Inet4Address` представляет традиционные адреса IPv4, а класс `Inet6Address` инкапсулирует адреса IPv6 нового стиля. Поскольку оба являются подклассами класса `InetAddress`, ссылки на класс `InetAddress` могут указывать и на них. Это единственный способ, благодаря которому удалось добавить в Java функциональные возможности IPv6, не нарушая работы существующего кода и не добавляя большого количества новых классов. В большинстве случаев вы просто можете использовать класс `InetAddress`, работая с IP-адресами, поскольку этот класс приспособлен для обоих стилей.

Клиентские сокеты TCP/IP

Сокеты TCP/IP применяются для реализации надежных двунаправленных, постоянных соединений между точками — хостами в Интернете на основе потоков. Сокет может использоваться для подключения системы ввода-вывода Java к другим программам, которые могут находиться как на локальной машине, так и на любой другой машине в Интернете.

На заметку! Как правило, аплеты могут устанавливать сокетные соединения только с тем хостом, с которого они были загружены. Это ограничение введено в связи с тем, что было бы опасно для аплетов, загруженных через брандмауэр, иметь доступ к любой произвольной машине.

В Java существует два вида сокетов TCP — для серверов и клиентов. Класс `ServerSocket` является "слушателем", который ожидает подключения клиентов прежде, чем что-либо делать. Другими словами, класс `ServerSocket` предназначен для серверов. Класс `Socket` применяется для клиентов. Он разработан так, чтобы соединяться с серверными сокетами и инициировать обмен по протоколу. Поскольку клиентские сокеты наиболее часто применяются в приложениях Java, их мы и рассмотрим здесь. В табл. 21.2 описаны два конструктора, используемые для создания клиентских сокетов.

Класс `Socket` определяет несколько методов экземпляра. Например, класс `Socket` может быть просмотрен в любое время на предмет извлечения информации об адресе и порте, ассоциированной с ним. Для этого применяются методы, перечисленные в табл. 21.3.

Таблица 21.2. Конструкторы класса Socket

Конструктор	Описание
`Socket(String имяХоста, int порт) throws UnknownHostException, IOException`	Создает сокет, подключенный к именованному хосту и порту
`Socket(InetAddress ipАдрес, int порт) throws IOException`	Создает сокет, используя ранее существующий объект класса `InetAddress` и порт

Таблица 21.3. Методы экземпляра класса Socket

Метод	Описание
`InetAddress getInetAddress()`	Возвращает объект класса `InetAddress`, ассоциированный с объектом класса `Socket`. В случае если сокет не подключен, возвращает значение `null`
`int getPort()`	Возвращает дистанционный порт, к которому подключен вызывающий объект класса `Socket`. Если сокет не подключен, возвращает значение `0`
`int getLocalPort()`	Возвращает локальный порт, к которому привязан вызывающий объект класса `Socket`. Если сокет не привязан, возвращает значение `-1`

Вы можете получить доступ к входному и выходному потокам, ассоциированным с классом `Socket`, с использованием методов `getInputStream()` и `getOuptutStream()`, которые описаны в табл. 21.4. Каждый из них может передать исключение `IOException`, если сокет стал недействительным из-за утери соединения. Эти потоки используются точно так же, как потоки ввода-вывода, рассмотренные в главе 19, для получения и приема данных.

Таблица 21.4. Методы доступа к входному и выходному потокам, связанным с классом Socket

Метод	Описание
`InputStream getInputStream() throws IOException`	Возвращает объект класса `InetAddress`, ассоциированный с вызывающим сокетом
`OutputStream getOutputStream() throws IOException`	Возвращает объект класса `OutputStream`, ассоциированный с вызывающим сокетом

Доступно также еще несколько других методов, включая метод `connect()`, позволяющий задать новое подключение; метод `isConnected()`, возвращающий значение `true`, если сокет подключен к серверу; метод `isBound()`, возвращающий значение `true`, если сокет привязан к адресу; и метод `isClosed()`, возвращающий значение `true`, когда сокет закрыт. Чтобы закрыть сокет, вызовите метод `close()`. Закрытие сокета приводит к закрытию также связанных с ним потоков ввода-вывода. Начиная с JDK 7 класс `Socket` реализует также интерфейс `AutoCloseable`. Это значит, что для управления сокетом вы можете использовать блок `try`-*с-ресурсами*.

Следующая программа представляет простой пример применения класса `Socket`. Она открывает соединение с портом "whois" (порт 43) на сервере InterNIC, посылает сокету аргументы командной строки, а затем выводит возвращенные данные. Сервер InterNIC пытается трактовать аргумент как зарегистри-

рованное доменное имя Интернета, а затем возвращает IP-адрес и контактную информацию для этого сайта.

```
// Демонстрация работы с сокетами.
import java.net.*;
import java.io.*;

class Whois {
    public static void main(String args[]) throws Exception {
        int c;

        // Создает сокетное соединение с internic.net, порт 43.
        Socket s = new Socket("whois.internic.net", 43);

        // Получает входной и выходной потоки.
        InputStream in = s.getInputStream();
        OutputStream out = s.getOutputStream();

        // Создать строку запроса.
        String str = (args.length == 0 ? "MHProfessional.com" : args[0])
                     + "\n";

        // Преобразует в байты.
        byte buf[] = str.getBytes();

        // Посылает запрос.
        out.write(buf);

        // Читает и отображает ответ.
        while ((c = in.read()) != -1) {
            System.out.print((char) c);
        }
        s.close();
    }
}
```

Если, к примеру, вы запросите информацию об адресе `MHProfessional.com`, то получите нечто вроде следующего.

```
Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

    Domain Name: MHPROFESSIONAL.COM
    Registrar: MELBOURNE IT, LTD. D/B/A INTERNET NAMES WORLDWIDE
    Whois Server: whois.melbourneit.com
    Referral URL: http://www.melbourneit.com
    Name Server: NS1.MHEDU.COM
    Name Server: NS2.MHEDU.COM
.
.
.
```

Вот как работает эта программа. Сначала создается объект класса `Socket`, задающий имя хоста `"whois.internic.net"` и номер порта 43 (`internic.net` — это веб-сайт InterNIC, обрабатывающий запросы whois; порт 43 предназначен именно для этой службы). Затем и входной, и выходной потоки открываются в сокете. Далее создается строка, содержащая имя веб-сайта, информацию о котором вы хотите получить. В данном случае, если никакой сайт не указан в командной

строке, используется "MHProfessional.com". Строка преобразуется в байтовый массив и отправляется в сокет. После этого ответ читается из сокета, и результат отображается на экране. И наконец, сокет закрывается, что в результате приводит к закрытию потоков ввода-вывода. В приведенном примере сокет был закрыт вручную при вызове метода close(). Если вы используете комплект JDK 7, то можете использовать блок try-*с-ресурсами* для автоматического закрытия сокета. Вот, например, еще один способ написать метод main() предыдущей программы.

```
// Использование блока try-с-ресурсами для закрытия сокета.
public static void main(String args[]) throws Exception {
    int c;

    // Создает сокетное соединение с internic.net, порт 43.
    // Этим сокетом управляет блок try-с-ресурсами.
    try ( Socket s = new Socket("whois.internic.net", 43) ) {

        // Получает входной и выходной потоки.
        InputStream in = s.getInputStream();
        OutputStream out = s.getOutputStream();

        // Создать строку запроса.
        String str = (args.length == 0 ? "MHProfessional.com" : args[0])
                     + "\n";
        // Преобразует в байты.
        byte buf[] = str.getBytes();

        // Посылает запрос.
        out.write(buf);

        // Читает и отображает ответ.
        while ((c = in.read()) != -1) {
            System.out.print((char) c);
        }
    }

    // Теперь сокет закрыт.
}
```

В этой версии сокет автоматически закрывается в конце блока try.

Таким образом, пример не будет работать с версиями Java до JDK 7, а чтобы четко проиллюстрировать, когда сетевой ресурс может быть закрыт, в последующих примерах будет продолжено использование явного вызова метода close(). Однако в собственном коде следует обратить внимание на использование автоматического управления ресурсами, так как это обеспечивает более гибкий подход. Еще один момент: в этой версии исключения все еще передаются из функции main(), но они могли бы быть обработаны при добавлении директивы catch в конец блока try-*с-ресурсами*.

На заметку! Для простоты в примерах этой главы все исключения передаются из функции main(). Это позволяет проще проиллюстрировать логику сетевого кода. Однако в реальном коде вы обычно будете должны обработать исключения соответствующим способом.

Класс URL

Предыдущий пример не очень вразумителен, поскольку в настоящее время Интернет не ассоциируется со старыми протоколами, такими как whois, finger и FTP. Здесь царствует WWW – всемирная паутина (World Wide Web). Веб – это сла-

бо связанная коллекция высокоуровневых протоколов и форматов файлов, унифицированным образом используемых веб-браузерами. Одним из наиболее важных аспектов веб является то, что Тим Бернерс-Ли (Tim Berners-Lea) предложил масштабируемый способ нахождения всех ресурсов в Интернете. Как только вы можете однозначно именовать что-либо, это становится очень мощной парадигмой. Именно это и делает *унифицированный локатор ресурсов* (Uniform Resource Locator – URL).

URL обеспечивает довольно четкую форму уникальной идентификации адресной информации в веб. Внутри библиотеки классов Java класс `URL` представляет простой согласованный программный интерфейс для доступа к информации по всей сети при помощи URL.

Все URL совместно используют один и тот же базовый формат, хотя и допускающий некоторые вариации. Приведем два примера: `http://www.MHProfessional.com/` и `http://www.MHProfessional.com:80/index.htm`. Спецификация URL основана на четырех компонентах. Первый – используемый протокол, отделяемый от остальной части локатора двоеточием (`:`). Распространенными протоколами являются HTTP, FTP, gopher и file, хотя в наши дни почти все осуществляется через протокол HTTP (фактически большинство браузеров корректно работает, даже если вы исключите из спецификации URL фрагмент `"http://"`). Второй компонент – имя хоста или IP-адрес, используемый хостом; он отделяется слева двойным слешем (`//`), а справа – слешем (`/`) или, что необязательно, двоеточием (`:`). Третий компонент – номер порта; это необязательный параметр, отделяемый слева от имени хоста двоеточием, а справа – слешем (`/`). (Если порт 80 является портом по умолчанию для протокола HTTP, то указывать `":80"` излишне.) Четвертый компонент – действительный путь к файлу. Большинство серверов HTTP добавляет имя файла `index.html` или `index.htm` к URL, которые указывают непосредственно на какой-то каталог. Таким образом, `http://www.MHProfessional.com/` – это то же самое, что и адрес `http://www.MHProfessional.com/index.htm`.

Класс Java `URL` имеет несколько конструкторов, и каждый из них может передать исключение `MalformedURLException`. Одна из часто используемых форм определяет URL в виде строки, идентичной тому, что вы видите в браузере.

```
URL(String спецификаторURL) throws MalformedURLException
```

Следующие две формы конструктора позволяют вам разделить URL на части-компоненты.

```
URL(String имяПротокола, String имяХоста, int порт, String путь)
    throws MalformedURLException
URL(String имяПротокола, String имяХоста, String путь)
    throws MalformedURLException
```

Другой часто используемый конструктор позволяет указать существующий URL в качестве ссылочного контекста, а затем создать из этого контекста новый URL. Хотя это звучит несколько запутано, на самом деле это очень просто и удобно.

```
URL(URL объектURL, String спецификаторURL) throws MalformedURLException
```

Следующий пример создает URL страницы статей `HerbSchildt.com`, а затем просматривает его свойства.

```
// Демонстрация применения URL.
import java.net.*;
class URLDemo {
    public static void main(String args[]) throws MalformedURLException {
        URL hp = new URL("http://www.HerbSchildt.com/Articles");

        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("Целиком: " + hp.toExternalForm());
    }
}
```

Запустив это, вы получите следующее.

```
Protocol: http
Port: -1
Host: www.HerbSchildt.com
File: /Articles
Ext:http://www.HerbSchildt.com/Articles
```

Обратите внимание на порт -1; это означает, что порт явно не установлен. Передав объект URL, вы можете извлечь ассоциированные с ним данные. Чтобы получить доступ к действительным битам или информации по URL, создайте из него объект класса URLConnection, используя его метод openConnection(), как показано ниже.

```
urlc = url.openConnection()
```

Метод openaConnection() имеет следующую общую форму.

```
URLConnection openConnection() throws IOException
```

Он возвращает объект класса URLConnection, ассоциированный с вызывающим объектом класса URL. Обратите внимание: он может передать исключение IOException.

Класс URLConnection

Класс URLConnection — это класс общего назначения, предназначенный для доступа к атрибутам удаленного ресурса. Однажды установив соединение с удаленным сервером, вы можете использовать класс URLConnection для просмотра свойств удаленного объекта, прежде чем транспортировать его локально. Эти атрибуты представлены в спецификации протокола HTTP и, как таковые, имеют смысл только для объектов URL, использующих протокол HTTP.

Класс URLConnection определяет несколько методов. Некоторые из них перечислены в табл. 21.5.

Таблица 21.5. Некоторые методы класса URLConnection

Метод	Описание
int getContentLength()	Возвращает размер в байтах содержимого, связанного с ресурсом. Если размер недоступен, возвращает значение -1
long getContentLengthLong()	Возвращает размер в байтах содержимого, связанного с ресурсом. Если размер недоступен, возвращает значение -1 (Добавлено в JDK 7)
String getContentType()	Возвращает тип содержимого, найденного в ресурсе. Это значение поля заголовка content-type. Возвращает значение null, если тип содержимого недоступен
long getDate()	Возвращает время и дату ответа, представленное в миллисекундах, прошедших с 1 января 1970 г.
long getExpiration()	Возвращает время и дату устаревания ресурса, представленное в миллисекундах, прошедших с 1 января 1970 г. Если дата устаревания недоступна, возвращается нуль

Окончание табл. 21.5

Метод	Описание
`String getHeaderField(int` *индекс*`)`	Возвращает значение заголовочного поля по индексу *индекс*. (Индексы полей заголовка нумеруются начиная с 0.) Возвращает значение `null`, если значение *индекс* превышает количество полей
`String getHeaderField(String` *имяПоля*`)`	Возвращает значение заголовочного поля, имя которого указано в *имяПоля*. Возвращает значение `null`, если указанное поле не найдено
`String getHeaderFieldKey(int` *индекс*`)`	Возвращает ключ заголовочного поля по индексу *индекс*. (Индексы полей заголовка нумеруются начиная с 0.) Возвращает значение `null`, если значение *индекс* превышает количество полей
`Map<String, List<String>> getHeaderFields()`	Возвращает карту, содержащую все заголовочные поля вместе с их значениями
`long getLastModified()`	Возвращает время и дату последней модификации ресурса, представленные в миллисекундах, прошедших после 1 января 1970 г. Если эта информация недоступна, возвращается нуль
`InputStream getInputStream() throws IOException`	Возвращает объект класса `InputStream`, привязанный к ресурсу. Данный поток может использоваться для получения содержимого ресурса

Обратите внимание на то, что класс `URLConnection` определяет несколько методов, управляющих заголовочной информацией. Заголовок состоит из пар ключей и значений, представленных в виде строк. Используя метод `getHeaderField()`, вы можете получить значение, ассоциированное с ключом заголовка. Вызывая метод `getHeaderField()`, можно получить карту, содержащую все заголовки. Несколько стандартных заголовочных полей доступны непосредственно через такие методы, как `getDate()` и `getContentType()`.

Следующий пример создает объект класса `URLConnection`, используя метод `openConnection()` объекта класса `URL`, а затем применяет его для проверки свойств и содержимого документа.

```
// Демонстрация применения 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("Нет информации о дате.");
        else
            System.out.println("Дата: " + new Date(d));

        // получить тип содержимого
        System.out.println("Тип содержимого: " + hpCon.getContentType());

        // получить дату устаревания
        d = hpCon.getExpiration();
```

```
        if(d==0)
            System.out.println("Нет информации о сроке действия.");
        else
            System.out.println("Устареет: " + new Date(d));

        // получить дату последней модификации
        d = hpCon.getLastModified();
        if(d==0)
            System.out.println(
                        "Нет информации о дате последней модификации.");
        else
            System.out.println(
                        "Дата последней модификации: " + new Date(d));

        // получить длину содержимого
        long len = hpCon.getContentLengthLong();
        if(len == -1)
            System.out.println("Длина содержимого недоступна.");
        else
            System.out.println("Длина содержимого: " + 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.");
        }
    }
}
```

Эта программа устанавливает соединение HTTP с сервером `www.internic.net` через порт 80. Затем она отображает несколько заголовочных значений и извлекает содержимое. Приведем первые строки вывода (точное их содержание будет меняться со временем).

```
Дата: Mon Oct 04 15:53:24 CDT 2010
Тип содержимого: text/html; charset=UTF-8
Нет информации о сроке действия.
Дата последней модификации: Thu Sep 24 15:22:52 CDT 2009
Длина содержимого: 7316
=== Содержимое ===
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>InterNIC | The Internet's Network Information Center</title>
.
.
.
```

Класс `HttpURLConnection`

Java предлагает подкласс класса `URLConnection`, обеспечивающий поддержку соединений HTTP. Этот класс называется `HttpURLConnection`. Вы получаете объект класса `HttpURLConnection` точно так же, как было показано, — вызовом метода `openConnection()` объекта класса `URL`, но результат следует приводить к классу `HttpURLConnection`. (Конечно, необходимо убедиться в том, что вы

действительно открыли соединение HTTP.) Получив ссылку на объект класса HttpURLConnection, вы можете вызывать любые его методы, унаследованные от класса URLConnection. Вы также можете использовать любые методы, определенные в классе HttpURLConnection. Некоторые методы перечислены в табл. 21.6.

Таблица 21.6. Некоторые методы класса HttpURLConnection

Метод	Описание
`static boolean getFollowRedirects()`	Возвращает значение `true`, если автоматически следует перенаправление, и значение `false` — в противном случае
`String getRequestMethod()`	Возвращает строковое представление метода выполнения запроса. По умолчанию используется метод GET. Доступны другие методы, такие как POST
`int getResponseCode() throws IOException`	Возвращает код ответа HTTP. Если код ответа не может быть получен, возвращается значение `-1`. При разрыве соединения передается исключение `IOException`
`String getResponseMessage() throws IOException`	Возвращает сообщение ответа, ассоциированное с кодом ответа. Если никакого сообщения недоступно, возвращает значение `null`
`static void setFollowRedirects(boolean как)`	Если параметр *как* содержит значение `true`, значит, перенаправление осуществляется автоматически. Если же он содержит значение `false`, значит, этого не происходит. По умолчанию перенаправление осуществляется автоматически
`void setRequestMethod(String как) throws ProtocolException`	Устанавливает метод, которым выполняются запросы HTTP, в соответствии с указанным в параметре *как*. По умолчанию принят метод GET, но доступны также другие варианты, такие как POST. Если в параметре *как* указано неправильное значение, передается исключение `ProtocolException`

Вследующейпрограммедемонстрируетсяработасклассом`HttpURLConnection`. Сначала она устанавливает соединение с сайтом `www.google.com`. Затем отображает метод запроса, код ответа и сообщение ответа. И наконец, отображает ключи и значения в заголовке ответа.

```
// Демонстрация применения 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();

        // Отображение метода запроса.
        System.out.println("Метод запроса: " + hpCon.getRequestMethod());

        // Отображение кода ответа.
        System.out.println("Код ответа: " + hpCon.getResponseCode());

        // Отображение сообщения ответа.
        System.out.println("Сообщение ответа: " +
```

```
                                hpCon.getResponseMessage());

        // Получить список полей заголовка и набор его ключей.
        Map<String, List<String>> hdrMap = hpCon.getHeaderFields();
        Set<String> hdrField = hdrMap.keySet();
        System.out.println("\nЗдесь следует заголовок:");

        // Отобразить все ключи и значения заголовка.
        for(String k : hdrField) {
            System.out.println("Ключ: " + k + " Значение: " +
                               hdrMap.get(k));
        }
    }
}
```

Вывод этой программы показан ниже (разумеется, точный ответ, возвращенный сайтом `www.google.com`, будет меняться с течением времени).

```
Метод запроса: GET
Код ответа: 200
Сообщение ответа: OK
Здесь следует заголовок:
Ключ: null Value: [HTTP/1.1 200 OK]
Ключ: Date Value: [Mon, 04 Oct 2010 21:11:53 GMT]
Ключ: Transfer-Encoding Value: [chunked]
Ключ: Expires Value: [-1]
Ключ: X-XSS-Protection Value: [1; mode=block]
Ключ: Set-Cookie Value: [NID=39=uAS1-
DdTfLelHcxkEiRy7xNtExX3zJaKS9mjdTy8_XejjBkpjWvcqyMXgC4Ha4VT_5IZN2pnxsl
oo-NlGHvcK0AIqXPhFcnCdlR1Ww4WgbiY7KrthNXCQxfXbHJwNgue; expires=Tue,
05-Apr-201121:11:53 GMT; path=/; domain=.google.com; HttpOnly,
PREF=ID=6644372b1f96120c:TM=1286226713:LM=1286226713:S=
iNeZU0xRTrGPxg2K;expires=Wed, 03-Oct-2012 21:11:53 GMT;
path=/; domain=.google.com]
Ключ: Content-Type Value: [text/html; charset=ISO-8859-1]
Ключ: Server Value: [gws]
Ключ: Cache-Control Value: [private, max-age=0]
```

Обратите внимание на то, как отображаются ключи и значения заголовка. Карта ключей и заголовков получается вызовом метода `getHeaderFields()` (который унаследован от класса `URLConnection`). Затем набор ключей заголовка извлекается вызовом метода `keySet()` карты. Далее осуществляется перебор всего набора при помощи стиля "for-each" цикла `for`. Значение, ассоциированное с каждым ключом, получается из карты вызовом метода `get()`.

Класс `URI`

Класс `URI` инкапсулирует *универсальный идентификатор ресурса* (Uniform Resource Identifier – URI). URI очень похож на URL. На самом деле URL представляет собой подмножество URI. URI предоставляет стандартный способ идентификации ресурсов. URL также описывает доступ к ресурсу.

Файлы cookie

Пакет `java.net` включает классы и интерфейсы, помогающие управлять файлами cookie, которые могут использоваться для создания сеансов HTTP с поддержкой состояния (в противоположность таковым без поддержки состояния). К таким классам относятся `CookieHandler`, `CookieManager` и `HttpCookie`, а к интерфейсам –

CookiePolicy и CookieStore. Все, кроме класса CookieHandler, были добавлены в Java SE 6. (Класс CookieHandler появился в JDK 5.) Создание сеанса HTTP с поддержкой состояния выходит за рамки настоящей книги.

На заметку! Информацию о применении файлов cookie с сервлетами см. в главе 32.

Серверные сокеты TCP/IP

Как уже упоминалось, в языке Java имеются различные классы сокетов, которые должны применяться для создания серверных приложений. Класс ServerSocket используется для создания серверов, которые прослушивают обращения как локальных, так и удаленных клиентских программ, желающих установить соединения с ними через открытые порты. Класс ServerSocket довольно-таки сильно отличается от обычных классов Socket. Когда вы создаете объект класса ServerSocket, он регистрирует себя в системе в качестве заинтересованного в клиентских соединениях. Конструкторы класса ServerSocket отражают номер порта, через который вы хотите принимать соединения, а также (необязательно) длину очереди для данного порта. Длина очереди сообщает системе о том, сколько клиентских соединений можно удерживать, прежде чем начать просто отклонять попытки подключения. По умолчанию установлено 50. При определенных условиях конструкторы могут передать исключение IOException. Конструкторы этого класса описаны в табл. 21.7.

Таблица 21.7. Конструкторы класса ServerSocket

Конструктор	Описание
`ServerSocket(int` *порт*`) throws IOException`	Создает серверный сокет на указанном порте с длиной очереди 50
`ServerSocket(int` *порт*`, int` *максОчередь*`) throws IOException`	Создает серверный сокет на указанном порте с максимальной длиной очереди в параметре *максОчередь*
`ServerSocket(int` *порт*`, int` *максОчередь*`, InetAddress` *локальныйАдрес*`) throws IOException`	Создает серверный сокет на указанном порте с максимальной длиной очереди в параметре *максОчередь*. На групповом хосте *локальныйАдрес* указывает IP-адрес, к которому привязан сокет

Класс ServerSocket включает метод по имени accept(), представляющий собой блокирующий вызов, который будет ожидать от клиента инициации соединений, а затем возвратит обычный объект класса Socket, который далее может служить для взаимодействия с клиентом.

Дейтаграммы

Сетевое взаимодействие в стиле TCP/IP подходит для большинства сетевых нужд. Оно обеспечивает сериализуемые, предсказуемые и надежные потоки пакетов данных. Тем не менее это обходится далеко не даром. Протокол TCP включает множество сложных алгоритмов управления потоками в нагруженных сетях, а также самые пессимистические предположения относительно утери пакетов. Это порождает в некоторой степени неэффективный способ транспортировки данных. В качестве альтернативы можно использовать дейтаграммы.

Дейтаграммы (datagramms) — это порции информации, передаваемые между машинами. В некотором отношении они подобны сильным броскам тренированного, но подслеповатого кетчера в сторону третьего бейсмена. Как только дейта-

грамма запущена в нужном направлении, нет никаких гарантий, что она достигнет цели или кто-нибудь окажется на месте, чтобы ее подхватить. Точно так же, когда дейтаграмма принимается, нет никакой гарантии, что она не была повреждена в пути или что ее отправитель все еще ожидает ответа.

Java реализует дейтаграммы поверх протокола UDP, используя для этого два класса: `DatagramPacket` (контейнер данных) и класс `DatagramSocket` (механизм, используемый для обслуживания класса `DatagramPacket`). Рассмотрим их более подробно.

Класс `DatagramSocket`

Класс `DatagramSocket` определяет четыре открытых конструктора.

```
DatagramSocket() throws SocketException
DatagramSocket(int порт) throws SocketException
DatagramSocket(int порт, InetAddress ipАдрес) throws SocketException
DatagramSocket(SocketAddress адрес) throws SocketException
```

Первый конструктор создает объект класса `DatagramSocket`, связанный с любым незанятым портом локального компьютера. Второй — объект класса `DatagramSocket`, связанный с портом, указанным в `port`. Третий создает объект класса `DatagramSocket`, связанный с указанным портом и объектом класса `InetAddress`. Четвертый — объект класса `DatagramSocket`, связанный с заданным `SocketAddress`. Класс `SocketAddress` — это абстрактный класс, реализуемый конкретным классом `InetSocketAddress`, который инкапсулирует IP-адрес с номером порта. Все конструкторы могут передать исключение `SocketException` в случае возникновения ошибок во время создания сокета.

Класс `DatagramSocket` определяет множество методов. Два наиболее важных из них — это методы `send()` и `receive()`, которые представлены ниже.

```
void send(DatagramPacket пакет) throws IOException
void receive(DatagramPacket пакет) throws IOException
```

Метод `send()` отправляет порту пакет, указанный в параметре *пакет*. Метод `receive()` ожидает приема через порт пакета, указанного в параметре *пакет*, и возвращает результат.

Класс `DatagramSocket` определяет также метод `close()`, который закрывает сокет. Начиная с JDK 7 класс `DatagramSocket` реализует интерфейс `AutoCloseable`, что позволяет управлять им блоку `try`-*с-ресурсами*.

Другие методы предоставляют вам доступ к различным атрибутам, связанным с классом `DatagramSocket`. Эти методы перечислены в табл. 21.8.

Таблица 21.8. Методы класса `DatagramSocket`

Метод	Описание
`InetAddress getInetAddress()`	Если сокет подключен, возвращается адрес. В противном случае возвращается значение `null`
`int getLocalPort()`	Возвращает номер локального порта
`int getPort()`	Возвращает номер порта, к которому подключен сокет. Возвращает значение `-1`, если сокет не подключен ни к какому порту
`boolean isBound()`	Возвращает значение `true`, если сокет привязан к адресу, в противном случае — значение `false`
`boolean isConnected()`	Возвращает значение `true`, если сокет подключен к серверу, в противном случае — значение `false`
`void setSoTimeout(int` *миллисекунд*`) throws SocketException`	Устанавливает период ожидания в миллисекундах, переданный параметром *миллисекунд*

Класс DatagramPacket

Класс DatagramPacket определяет множество конструкторов. Вот четыре из них.

```
DatagramPacket(byte данные[ ], int размер)
DatagramPacket(byte данные[ ], int смещение, int размер)
DatagramPacket(byte данные[ ], int размер, InetAddress ipAddress, int порт)
DatagramPacket(byte данные[ ], int смещение, int размер, InetAddress
ipAdpec, int порт)
```

Первый конструктор определяет буфер, который будет принимать данные, и размер пакета. Он используется для приема данных через класс DatagramSocket. Второй конструктор позволяет вам указать смещение в буфере, куда должны быть помещены данные. Третий указывает целевой адрес и порт, используемые классом DatagramSocket для определения того, куда данные пакета будут отправлены. Четвертый передает пакеты, начиная с указанного смещения в данных. Первые два конструктора следует воспринимать как построение "ящика", а вторые два — как "начинку" и адрес на конверте.

Класс DatagramPacket определяет несколько методов, включая представленные здесь, которые предоставляют доступ к адресу и номеру порта пакета, наряду с базовыми данными и их длиной (табл. 21.9). В общем случае методы get используются в принимаемых пакетах, а методы set — в отправляемых.

Таблица 21.9. Методы класса 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 приема
`int getLength()`	Возвращает длину корректных данных, содержащихся в байтовом массиве, который должен быть возвращен из метода `getData()`. Это может не совпадать с полной длиной байтового массива
`int getOffset()`	Возвращает начальный индекс данных
`int getPort()`	Возвращает номер порта
`void setAddress(InetAddress ipAdpec)`	Устанавливает адрес, по которому отправляется пакет. Адрес указывается параметром *ipAdpec*
`void setData(byte[] данные)`	Устанавливает данные в *данные*, смещение — в нуль, а длину — в количество байтов *данные*
`void setData(byte[] данные, int индекс, int размер)`	Устанавливает данные в *данные*, смещение — в *индекс*, а длину — в *размер*
`void setLength(int размер)`	Устанавливает длину пакета в *размер*
`void setPort(int порт)`	Устанавливает порт в *порт*

Пример работы с дейтаграммами

В следующем примере реализуется очень простое сетевое взаимодействие: клиент и сервер. Сообщения вводятся в окне на сервере и передаются по сети на сторону клиента, где и отображаются.

```
// Демонстрация применения 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 завершил работу.");
                    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();
        }
    }
}
```

Этот пример программы ограничен конструктором класса `DtagramSocket` для запуска между портами локальной машины. Чтобы использовать программы, выполните в одном окне такую команду.

```
java WriteServer
```

Это будет клиент. Затем в другом окне запустите следующую команду.

```
java WriteServer 1
```

Это будет сервер. Все, что вы введете в окне сервера, после получения символа перевода строки будет отправлено в клиентское окно.

ГЛАВА 22

Класс Applet

В настоящей главе рассматривается класс Applet, который обеспечивает основу для создания аплетов. Этот класс содержится в пакете java.applet. Он включает несколько методов, обеспечивающих подробный контроль над выполнением аплета. Кроме того, пакет java.applet определяет три интерфейса — AppletContext, AudioClip и AppletStub.

Два типа аплетов

Важно подчеркнуть, что существует два типа аплетов. Первый основан непосредственно на классе Applet, описанном в настоящей главе. Эти аплеты используют библиотеку Abstract Window Toolkit (AWT) для предоставления пользовательского графического интерфейса (если вообще его используют). Этот тип аплетов доступен с самого начала существования Java.

Второй тип аплетов основан на классе JApplet библиотеки Swing. Аплеты Swing используют классы библиотеки Swing для построения графического интерфейса пользователя. Библиотека Swing предлагает более богатый и зачастую более легкий в применении пользовательский интерфейс, чем библиотека AWT. Поэтому аплеты на основе библиотеки Swing в настоящее время наиболее популярны. Однако традиционные аплеты на основе библиотеки AWT все еще применяются, особенно когда требуется построить очень простой пользовательский интерфейс. А потому и аплеты на основе библиотеки AWT, и аплеты на основе библиотеки Swing совершенно законны.

Поскольку класс JApplet происходит от класса Applet, все его средства также доступны в классе JApplet, и большая часть материала этой главы относится к обоим типам аплетов. А потому, даже если вас интересуют только аплеты Swing, приведенная в этой главе информация все равно будет полезна и необходима. Однако следует понимать, что при создании аплетов на основе библиотеки Swing существуют некоторые дополнительные ограничения, которые будут описаны далее в книге, когда пойдет речь о библиотеке Swing.

На заметку! Информация о построении аплетов с использованием библиотеки Swing приведена в главе 30.

Основы аплетов

В главе 13 были рассмотрены общая форма аплета, а также действия, необходимые для его компиляции и запуска. Начнем с пересмотра этой информации.

Все классы аплетов являются подклассами (прямыми или непрямыми) класса `Applet`. Аплеты не являются самостоятельными программами, они выполняются либо внутри веб-браузера, либо при помощи средства просмотра аплетов. Иллюстрации, которые приведены в этой главе, были созданы с применением стандартного средства просмотра аплетов под названием `appletviewer`, входящего в комплект JDK. Однако вы можете использовать любой другой браузер или средство просмотра по своему вкусу.

Выполнение аплета начинается не с метода `main()`. На самом деле лишь немногие аплеты имеют методы `main()`. Вместо этого выполнение аплета запускается и управляется при помощи совершенно другого механизма, который будет описан ниже. Вывод окна вашего аплета осуществляется не при помощи метода `System.out.println()`. Вместо этого в аплетах не на базе библиотеки Swing вывод осуществляется различными методами библиотеки AWT, такими как метод `drawString()`, который направляет строку в точку с указанными координатами X,Y. Ввод также осуществляется иначе, чем в консольных приложениях. (Помните, что аплеты на основе библиотеки Swing используют классы библиотеки Swing для обработки взаимодействия с пользователем, и они также будут описаны далее в настоящей книге.)

Прежде чем использовать аплет, следует выбрать стратегию его развертывания. Есть два основных подхода. Первый подразумевает использование *протокола запуска сети Java* (Java Network Launch Protocol – JNLP). Этот подход обеспечивает больше гибкости, особенно в отношении улучшенных приложений Интернета. Для реальных аплетов, которые вы создаете, протокол JNLP зачастую является наилучшим выбором. Однако подробное обсуждение протокола JNLP выходит за рамки данной книги. (Более подробная информация о протоколе JNLP приведена в документации комплекта JDK.) К счастью, протокол JNLP не обязателен для примеров аплетов, представленных здесь.

Второй подход развертывания аплета подразумевает его определение непосредственно в файле HTML, без использования протокола JNLP. Это первоначальный способ запуска аплетов, который использовался Java с момента создания и до сих пор применяется для простых аплетов. Кроме того, благодаря унаследованной простоте, этот подход применяется для примеров аплетов, описанных в данной книге. На момент написания книги корпорация Oracle рекомендовала дескриптор `APPLET`, поэтому в данной книге используется именно этот дескриптор. (В настоящее время применение дескриптора `APPLET` не рекомендуется спецификацией HTML. Альтернатива – дескриптор `OBJECT`. За последними рекомендациями в этом отношении имеет смысл обратиться к документации комплекта JDK.) Когда в файле HTML встречается дескриптор `APPLET`, веб-браузер с поддержкой Java выполняет указанный аплет.

Использование дескриптора `APPLET` предоставляет еще одно преимущество при разработке аплетов, поскольку он позволяет легко просматривать и проверять аплет. Для этого просто включите комментарий в заголовок файла вашего исходного кода Java, который содержит дескриптор `APPLET`. Таким образом, код будет документирован конструкциями HTML, необходимыми вашему аплету, и вы сможете проверить скомпилированный аплет, запустив средство просмотра аплетов с заданным файлом исходного кода Java. Вот пример такого комментария.

```
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```

Этот комментарий включает дескриптор `APPLET`, который запустит аплет по имени `MyApplet` в окне размером 200 пикселей в ширину и 60 пикселей в высоту.

Поскольку включение команды APPLET облегчает проверку аплетов, все аплеты, показанные в этой книге, будут содержать соответствующий дескриптор APPLET, помещенный в комментарий.

Класс Applet

Этот класс определяет методы, показанные в табл. 22.1. Класс Applet обеспечивает всю необходимую поддержку для выполнения аплета, такую как его запуск и останов. Кроме того, он предоставляет методы для загрузки и отображения графических образов, а также для загрузки и воспроизведения аудиоклипов. Класс Applet расширяет класс Panel библиотеки AWT. В свою очередь, класс Panel расширяет класс Container, который расширяет класс Component. Все эти классы обеспечивают поддержку графического оконного интерфейса на базе Java.

Таблица 22.1. Методы, определенные в классе Applet

Метод	Описание
`void destroy()`	Вызывается браузером непосредственно перед уничтожением аплета. Ваш аплет переопределяет этот метод, если нуждается в выполнении некоторых действий по очистке перед уничтожением
`AccessibleContext getAccessibleContext()`	Возвращает контекст доступности для вызывающего объекта
`AppletContext getAppletContext()`	Возвращает контекст, ассоциированный с аплетом
`String getAppletInfo()`	Переопределенные версии этого метода должны возвращать строку, которая описывает аплет. Реализация по умолчанию возвращает значение `null`
`AudioClip getAudioClip(URL url)`	Возвращает объект интерфейса `AudioClip`, инкапсулирующий аудиоклип, находящийся в месте, заданном *`url`*
`AudioClip getAudioClip(URL url, String имяКлипа)`	Возвращает объект интерфейса `AudioClip`, инкапсулирующий аудиоклип, находящийся в месте, заданном *`url`*, и имеющий имя, указанное *`имяКлипа`*
`URL getCodeBase()`	Возвращает URL, ассоциированный с вызывающим аплетом
`URL getDocumentBase()`	Возвращает URL документа HTML, вызвавшего аплет
`Image getImage(URL url)`	Возвращает объект класса `Image`, инкапсулирующий графическое изображение, найденное в месте, указанном параметром *`url`*
`Image getImage(URL url, String имяИзображения)`	Возвращает объект класса `Image`, инкапсулирующий графическое изображение, найденное в месте, указанном параметром *`url`*, и имеющий имя, указанное параметром *`имяИзображения`*
`Locale getLocale()`	Возвращает объект класса `Locale`, используемый различными классами и методами, чувствительными к региональным данным
`String getParameter(String имяПараметра)`	Возвращает параметр *`имяПараметра`*. Если указанный параметр не найден, возвращается значение `null`

Окончание табл. 22.1

Метод	Описание
`String[] [] getParameterInfo()`	Метод, переопределяющий данный метод, должен возвращать строковую таблицу, описывающую параметры, распознаваемые аплетом. Каждое вхождение в таблице должно состоять из трех строк, содержащих имя параметра, описание его типа и/или диапазон, а также все необходимые пояснения. Реализация по умолчанию возвращает значение `null`
`void init()`	Вызывается при запуске выполнения аплета. Это — первый метод, вызываемый аплетом
`boolean isActive()`	Возвращает значение `true`, если аплет запущен. Возвращает значение `false`, если аплет был остановлен
`boolean isValidateRoot()`	Возврат значения `true` указывает, что аплет проверяет корень. (Добавлено в JDK 7)
`static final AudioClip newAudioClip(URL` *`url`*`)`	Возвращает объект интерфейса `AudioClip`, инкапсулирующий аудиоклип, который расположен в месте, указанном параметром *`url`*. Этот метод подобен методу `getAudioClip()`, за исключением того, что является статическим и может быть выполнен без объекта класса `Applet`
`void play(URL` *`url`*`)`	Если аудиоклип найден в месте, указанном *`url`*, он выполняется
`void play(URL` *`url`*`, String` *`имяКлипа`*`)`	Если аудиоклип по имени *`имяКлипа`* найден в месте, указанном *`url`*, он выполняется
`void resize(Dimension` *`разм`*`)`	Изменяет размер аплета согласно измерениям, указанным в *`разм`*. `Dimension` — это класс, находящийся внутри файла `java.awt`. Он содержит два целочисленных поля — `width` и `height`
`void resize(int` *`ширина`*`, int` *`высота`*`)`	Изменяет размер аплета согласно размерностям, указанным параметрами *`ширина`* и *`высота`*
`final void setStub(AppletStub` *`объектЗагл`*`)`	Делает *`объектЗагл`* заглушкой (stub) для аплета. Этот метод применяется исполняющей системой и никогда не вызывается непосредственно вашим аплетом. *Заглушка* — это небольшой фрагмент кода, который обеспечивает связь вашего аплета с браузером
`void showStatus(String` *`строка`*`)`	Отображает *`строка`* в строке состояния окна браузера или средства просмотра аплетов. Если браузер не поддерживает окна состояния, то ничего не происходит
`void start()`	Вызывается браузером, когда аплет должен начать (или возобновить) выполнение. Автоматически вызывается после метода `init()` в начале работы аплета
`void stop()`	Вызывается браузером для приостановки выполнения аплета. Будучи остановленным, аплет перезапускается вызовом метода `start()`

Таким образом, класс `Applet` обеспечивает всю необходимую поддержку деятельности на основе окон. (Библиотека AWT будет подробно рассматриваться в последующих главах.)

Архитектура аплетов

Как правило, *аплет* (applet) — это оконная программа с графическим интерфейсом, и потому его архитектура отличается от архитектуры консольных программ, показанных в части I нашей книги. Если вы уже знакомы с программированием графического интерфейса, то при написании аплетов почувствуете себя буквально, как дома. Если же нет, то вы должны понимать некоторые концепции, которые будут описаны ниже.

Во-первых, аплеты управляются событиями. Хотя мы не будем рассматривать обработку событий до следующей главы, важно понимать в общем, как управляемая событиями архитектура влияет на дизайн аплетов. Аплет включает в себя набор служебных процедур, обрабатывающих прерывания. Вот как это работает. Аплет ожидает, пока не произойдет событие. Исполняющая система извещает аплет о событии, вызывая обработчик события, предоставленный аплетом. Как только это случилось, аплет должен предпринять соответствующие действия и немедленно возвратить управление. Это — важнейший момент. Большей частью, ваш аплет не должен входить в "режим" операций, в которых он будет удерживать управление в течение длительного периода. Вместо этого он должен выполнить определенные действия в ответ на события, а затем вернуть управление исполняющей системе. В тех ситуациях, когда вашему аплету нужно выполнять какое-то повторяющееся действие (например, отображать в окне прокручивающееся сообщение), следует запустить дополнительный поток выполнения. (Далее в настоящей главе вы увидите соответствующий пример.)

Во-вторых, пользователь инициирует взаимодействие с аплетом — и никакого пути в обход! Как вы знаете, в неоконных консольных программах, когда программе необходим ввод, она выводит приглашение пользователю, а затем вызывает некоторый метод ввода, например метод `readLine()`. В аплетах все работает не так. Вместо этого пользователь взаимодействует с аплетом, как он желает и когда хочет. Эти взаимодействия отсылаются аплету в виде событий, на которые тот должен отреагировать. Например, когда пользователь щелкает кнопкой мыши внутри окна аплета, передается событие щелчка. Если пользователь нажимает клавишу, когда окно аплета имеет фокус ввода, передается событие нажатия клавиши. Как вы увидите в последующих главах, аплеты могут содержать различные элементы управления, такие как экранные кнопки и флажки. Всякий раз, когда пользователь взаимодействует с одним из этих элементов управления, происходят события.

Архитектуру аплетов понять не так просто, как архитектуру консольных программ, однако Java облегчает это, насколько возможно. Если вы писали консольные программы под Windows (или другие операционные системы с графическим интерфейсом), то знаете, насколько устрашающе может выглядеть эта среда. К счастью, Java предоставляет в ваше распоряжение намного более ясный подход, которым можно овладеть гораздо быстрее.

Шаблон аплета

Все аплеты, кроме наиболее тривиальных, переопределяют методы, обеспечивающие базовые механизмы взаимодействия браузера или средства просмотра аплетов с самим аплетом и управляющие его выполнением. Четыре из этих методов — `init()`, `start()`, `stop()` и `destroy()` — применимы ко всем аплетам; они определены в классе `Applet`. Предусмотрены реализации по умолчанию всех этих методов. Однако в их переопределении не нуждаются лишь очень простые аплеты.

Аплеты на базе библиотеки AWT (вроде тех, что обсуждаются в настоящей главе) также зачастую переопределяют метод `paint()`, который определен в классе `Component` библиотеки AWT. Этот метод вызывается, когда вывод аплета должен быть заново отображен. (Аплеты на основе библиотеки Swing используют другой механизм для решения этой задачи.) Эти пять методов могут быть собраны в "шаблон", показанный ниже.

```
// Шаблон аплета.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends Applet {
    // Вызывается первым.
    public void init() {
        // инициализация
    }

    /* Вызывается вторым, после init(). Также вызывается
    при перезапуске аплета. */
    public void start() {
        // запускает или возобновляет выполнение
    }

    // Вызывается при останове аплета.
    public void stop() {
        // приостановка выполнения
    }

    /* Вызывается перед уничтожением аплета. Это - последний
    выполняемый метод. */
    public void destroy() {
        // выполняет завершающие действия
    }

    // Вызывается, когда окно аплета должно быть восстановлено.
    public void paint(Graphics g) {
        // перерисовка содержимого окна
    }
}
```

Хотя этот шаблон ничего и не делает, его можно откомпилировать и запустить. Будучи запущенным в средстве просмотра аплетов, он создает окно, показанное на рис. 22.1.

Инициализация и прекращение работы аплета

Важно понимать порядок вызова различных методов, показанных выше в шаблоне. Когда аплет стартует, вызываются по порядку следующие методы.

1. `init()`
2. `start()`
3. `paint()`

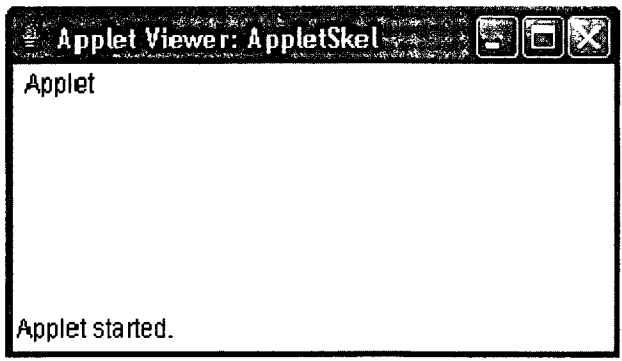


Рис. 22.1. Окно, отображаемое шаблоном аплета

Когда работа аплета прекращается, происходит следующая последовательность вызовов.

1. `stop()`
2. `destroy()`

Рассмотрим эти методы более подробно.

Метод `init()`

Это первый метод в цепочке вызовов. Именно в нем следует инициализировать переменные. На протяжении времени выполнения аплета этот метод вызывается лишь однажды.

Метод `start()`

После метода `init()` вызывается метод `start()`, который также вызывается для перезапуска аплета после его останова. В то время как метод `init()` вызывается лишь однажды, при первоначальной загрузке аплета, метод `start()` вызывается каждый раз, когда документ HTML, содержащий аплет, отображается на экране. Поэтому если пользователь покидает веб-страницу, а затем возвращается обратно, аплет каждый раз возобновляет свою работу в методе `start()`.

Метод `paint()`

Каждый раз, когда вывод аплета должен быть перерисован, вызывается метод `paint()`. Эта ситуация возникает в нескольких случаях. Например, окно, в котором выполняется аплет, может быть перекрыто другим окном, а затем вновь открыто. Или же окно аплета может быть минимизировано, а затем восстановлено. Метод `paint()` также вызывается в начале выполнения аплета. В любом случае всякий раз, когда нужно перерисовать аплет, вызывается его метод `paint()`. Метод `paint()` принимает один параметр типа `Graphics`. Этот параметр будет содержать графический контекст, описывающий графическую среду, в которой выполняется аплет. Этот контекст используется всякий раз, когда запрашивается вывод аплета.

Метод `stop()`

Этот метод вызывается, когда веб-браузер покидает документ HTML, содержащий аплет. Например, когда осуществляется переход на другую страницу. Когда

вызывается метод stop(), аплет, возможно, работает. Следует использовать метод stop() для приостановки потока, который не должен выполняться, когда аплет не видим. Вы можете перезапустить его, когда вызывается метод start(), при возврате пользователя на страницу.

Метод destroy()

Когда среда определяет, что ваш аплет должен быть полностью удален из памяти, вызывается метод destroy(). В этот момент следует освободить все ресурсы, которые мог использовать ваш аплет. Вызову метода destroy() всегда предшествует вызов метода stop().

Переопределение метода update()

В некоторых ситуациях вашему аплету может понадобиться переопределить другой метод, определенный в библиотеке AWT, — это метод update(). Этот метод вызывается, когда ваш аплет запрашивает перерисовку определенной части окна. Версия метода update() по умолчанию просто вызывает метод paint(). Однако вы можете переопределить метод update() так, чтобы он выполнял более тонкую перерисовку. Вообще, переопределение метода update() — это специализированная техника, которая не применяется во всех аплетах, и в примерах этой главы метод update() не переопределяется.

Простые методы отображения аплетов

Как мы уже упоминали, аплеты отображаются в окне, и аплеты на основе библиотеки AWT используют ее для выполнения ввода и вывода. Хотя в последующих главах мы будем подробно рассматривать методы, процедуры и приемы, используемые для полного управления оконной средой на базе библиотеки AWT, некоторые из них мы все же рассмотрим уже здесь, поскольку они понадобятся для разработки примеров аплетов. (Помните, что аплеты на основе библиотеки Swing будут рассматриваться в книге далее.)

Как было сказано в главе 13, для вывода строки в аплете используется метод drawstring() класса Graphics. Обычно этот метод вызывается внутри метода update() или paint(). Он имеет следующую общую форму.

```
void drawString(String сообщение, int x, int y)
```

Здесь *сообщение* — это строка, которая должна быть выведена начиная с точки с координатами *x*, *y*. В окне Java левый верхний угол имеет координаты 0, 0. Метод drawString() не распознает символы переноса строки. Если вы хотите начать вывод текста с новой строки, то должны сделать это явно, указав точные координаты X, Y точки, с которой нужно начать вывод этой новой строки текста (как увидите в последующих главах, существуют приемы, позволяющие облегчить этот процесс).

Чтобы установить цвет фона для окна аплета, используйте метод setBackground(). Чтобы установить цвет переднего плана (цвет вывода текста, например), используйте метод setForeground(). Эти методы определены в классе Component и имеют следующую общую форму.

```
void setBackground(Color новыйЦвет)
void setForeground(Color новыйЦвет)
```

Здесь параметр *новыйЦвет* определяет новый цвет. Класс `Color` определяет перечисленные ниже константы, которые могут быть использованы для указания цвета.

`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`	

Определены также версии этих констант в верхнем регистре. В следующем примере устанавливается зеленый цвет фона и красный цвет текста.

```
setBackground(Color.green);
setForeground(Color.red);
```

Подходящим для установки цветов фона и переднего плана является метод `init()`. Конечно, во время выполнения вашего аплета вы можете изменять эти цвета так часто, как хотите.

Получить текущие установки цветов переднего плана и текста можно вызовами методов `getBackground()` и `getForeground()` соответственно. Они также определены в классе `Component` и показаны ниже.

```
Color getBackground()
Color getForeground()
```

Рассмотрим пример очень простого аплета, который устанавливает в качестве цвета фона циан, а цвета переднего плана – красный, а затем отображает сообщение, иллюстрирующее порядок вызова методов `init()`, `start()` и `paint()` при запуске аплета.

```
/* Простой аплет, устанавливающий цвета фона
   и переднего плана и отображающий строку.
*/
import java.awt.*;
import java.applet.*;
/*

<applet code="Sample" width=300 height=50>
</applet>
*/
public class Sample extends Applet{
    String msg;

    // Установить цвета фона и переднего плана.
    public void init() {
        setBackground(Color.cyan);
        setForeground(Color.red);
        msg = "Inside init( ) --";
    }

    // Инициализировать отображаемую строку.
    public void start() {
        msg += " Inside start( ) --";
    }

    // Отобразить msg в окне аплета.
    public void paint(Graphics g) {
        msg += " Inside paint( ).";
```

```
        g.drawString(msg, 10, 30);
    }
}
```

Этот аплет создает окно, показанное на рис. 22.2.

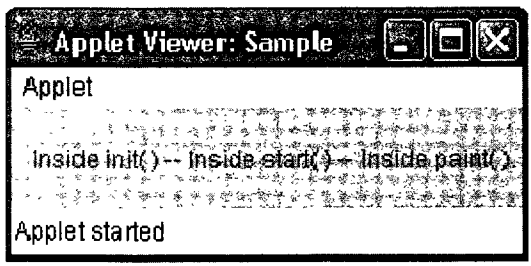


Рис. 22.2. Окно аплета, выводящего строку

Методы `stop()` и `destroy()` не переопределены, поскольку в таком простом аплете это не требуется.

Запрос перерисовки

Запомните главное правило: аплет пишет в свое окно только тогда, когда его методы `update()` или `paint()` вызываются библиотекой AWT. Отсюда интересный вопрос: как сам аплет может инициировать собственное обновление, когда изменяется его информация? Например, если аплет отображает двигающийся баннер, какой механизм заставит аплет обновлять окно при каждом шаге прокрутки этого баннера? Помните, что одним из наиболее важных ограничений, накладываемых на аплет, является то, что он должен быстро возвращать управление исполняющей системе. Например, он не может создавать циклы внутри метода `paint()`, которые будут непрерывно прокручивать баннер. Это помешало бы передаче управления обратно в библиотеку AWT. Имея такое ограничение, можно подумать, что вывод в окно вашего аплета, как минимум, существенно затруднен. К счастью, это не так. Всякий раз, когда ваш аплет нуждается в обновлении отображаемой в его окне информации, он просто вызывает метод `repaint()`.

Метод `repaint()` определен в библиотеке AWT. Он заставляет исполняющую систему библиотеки AWT осуществлять вызов метода `update()` вашего аплета, который в реализации по умолчанию обращается к методу `paint()`. Таким образом, чтобы другая часть вашего аплета могла выполнять вывод в его окно, просто сохраните вывод и вызовите метод `repaint()`. Затем библиотека AWT выполнит вызов метода `paint()`, который может отобразить сохраненную информацию. Например, если часть вашего аплета нуждается в выводе строки, она может сохранить ее в переменной класса `String`, а затем вызвать метод `repaint()`. Внутри метода `paint()` вы выведите строку с помощью метода `drawString()`.

Метод `repaint()` имеет четыре формы. Рассмотрим их по порядку. Следующая версия определяет область, подлежащую перерисовке.

```
void repaint(int слева, int сверху, int ширина, int высота)
```

Здесь координаты правого верхнего угла области указаны параметрами *слева* и *сверху*, а ширина и высота области — параметрами *ширина* и *высота*. Эти измерения указаны в пикселях. Вы экономите время, указывая область для перерисовки. Обновление окон обходится дорого в смысле затрат времени. Если вам нужно

обновить только небольшую часть окна, то эффективнее будет обновить только эту область, а не всю поверхность окна.

Вызов метода `repaint()` — это, по сути, запрос вашего аплета на скорейшее обновление. Но если ваша система работает медленно или занята, метод `update()` может и не вызваться немедленно. Множественные запросы на перерисовку, которые поступают за короткий период времени, могут быть собраны вместе, так что метод `update()` вызывается лишь время от времени. Во многих ситуациях это может представлять проблему, включая вывод анимации, когда существенно время обновления. Одним из решений этой проблемы может быть использование следующих форм метода `repaint()`.

```
void repaint(long макс3адержка)
void repaint(long макс3адержка, int x, int y, int ширина, int высота)
```

Здесь *макс3адержка* указывает максимальное количество миллисекунд, которые могут пройти до того, как будет вызван метод `update()`. Однако следует иметь в виду, что если время истечет прежде, чем системе удастся вызвать метод `update()`, этот метод не будет вызван. Никакого возвращаемого значения или передаваемого исключения нет, поэтому будьте осторожны.

На заметку! Существует возможность вывода в окно аплета методами, отличными от методов `paint()` или `update()`. Для вывода в окно такой метод должен получить графический контекст. Однако для большинства исключений лучше и проще осуществлять вывод окна через метод `paint()` и вызывать метод `repaint()` при изменении содержимого окна.

Простой аплет с баннером

Чтобы продемонстрировать метод `repaint()`, разработаем простой аплет для демонстрации баннера. Этот аплет будет прокручивать сообщение слева направо, через все окно аплета. Поскольку прокрутка окна — повторяющаяся задача, она будет выполняться в отдельном потоке, созданном аплетом при инициализации. Вот исходный текст нашего аплета баннера.

```
/* Простой аплет баннера.
Этот аплет создает поток, прокручивающий
сообщение, содержащееся в msg, справа налево
в поле окна аплета.
*/
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleBanner" width=300 height=50>
</applet>
*/

public class SimpleBanner extends Applet implements Runnable {
    String msg = " A Simple Moving Banner.";
    Thread t = null;
    int state;
    volatile boolean stopFlag;

    // Установить цвета и инициализировать поток.
    public void init() {
        setBackground(Color.cyan);
        setForeground(Color.red);
    }

    // Запустить поток
```

```
    public void start() {
        t = new Thread(this);
        stopFlag = false;
        t.start();
    }
    // Точка входа для потока, прокручивающего баннер.
    public void run() {

        // Снова отобразить баннер
        for( ; ; ) {
            try {
                repaint();
                Thread.sleep(250);
                if(stopFlag)
                    break;
            } catch(InterruptedException e) {}
        }
    }
    // Пауза в выводе баннера.
    public void stop() {
        stopFlag = true;
        t = null;
    }
    // Отображение баннера.
    public void paint(Graphics g) {
        char ch;

        ch = msg.charAt(0);
        msg = msg.substring(1, msg.length());
        msg += ch;

        g.drawString(msg, 50, 30);
    }
}
```

Окно этого аплета будет выглядеть, как показано на рис. 22.3.

Давайте рассмотрим внимательнее, как работает этот аплет. Для начала обратите внимание на то, что класс `SimpleBanner` расширяет класс `Applet`, как и следовало ожидать. Но, кроме того, он реализует интерфейс `Runnable`. Это необходимо, поскольку аплет создаст второй поток выполнения, который будет использован для прокрутки баннера. Внутри метода `init()` устанавливаются цвета фона и переднего плана аплета.

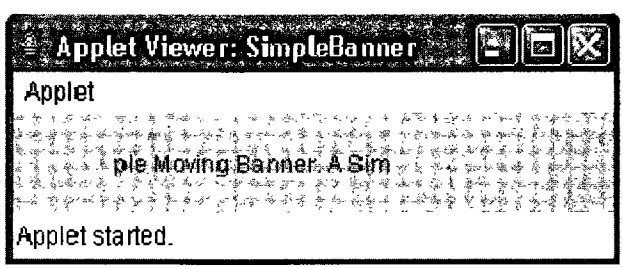


Рис. 22.3. Окно аплета с прокруткой сообщения

После инициализации исполняющая система вызывает метод `start()` для запуска выполнения аплета. Внутри метода `start()` создается новый поток выполнения и присваивается переменной `t` класса `Thread`. Затем в состояние `false`

устанавливается булева переменная stopFlag, которая управляет выполнением аплета. После этого поток запускается вызовом метода t.start(). Помните, что метод t.start() вызывает метод, определенный в классе Thread, т.е. начинает выполнять метод run(). Это не инициирует вызова версии метода start(), определенного в классе Applet. Это два отдельных метода.

Внутри метода run() вызывается метод repaint(). Это в конечном итоге приводит к вызову метода paint() и прокручиваемому отображению содержимого строки msg. Между итерациями метод run() "засыпает" на четверть секунды. В результате содержимое строки msg прокручивается слева направо в режиме непрерывного движения. Переменная stopFlag проверяется на каждой итерации. Когда она принимает значение true, метод run() прерывается.

Если браузер отображает аплет во время просмотра новой страницы, вызывается метод stop(), который устанавливает переменную stopFlag в состояние true, прерывая тем самым выполнение метода run(). Этот механизм служит для остановки потока, когда страница более не просматривается. Когда же аплет снова оказывается в поле зрения, вновь вызывается его метод start(), который запускает новый поток для прокрутки баннера.

Использование строки состояния

Кроме отображения информации в собственном окне, аплет может также выводить сообщения в строку состояния браузера или средства просмотра аплета, в котором он запущен. Чтобы сделать это, вызовите метод showStatus() со строкой, которую хотите отобразить. Строка состояния — хорошее место для того, чтобы дать пользователю представление о происходящем в аплете, показать параметры или вывести сообщения о некоторых типах ошибок. Строка состояния — отличный инструмент отладки, поскольку предоставляет простой способ вывода информации о вашем аплете.

В следующем аплете демонстрируется применение метода showStatus().

```
/ Использование окна состояния.
import java.awt.*;
import java.applet.*;
 *

<applet code="StatusWindow" width=300 height=50>
</applet>
*/

public class StatusWindow extends Applet {
    public void init() {
        setBackground(Color.cyan);
    }

    // Отобразить msg в окне аплета.
    public void paint(Graphics g) {
        g.drawString("This is in the applet window.", 10, 20);
        showStatus("This is shown in the status window.");
        // g.drawString("Это в окне аплета.", 10, 20);
        // showStatus("Это в строке состояния.");
    }
```

Окно аплета показано на рис. 22.4.

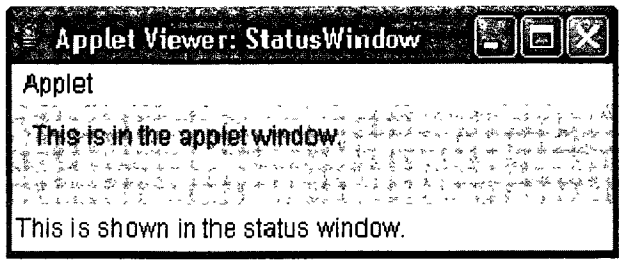


Рис. 22.4. Окно аплета, использующего строку состояния

Дескриптор HTML APPLET

Как уже упоминалось, на момент написания книги компания Oracle рекомендует дескриптор APPLET для запуска аплета вручную, когда протокол JNLP не используется. Средство просмотра аплетов выполнит каждый дескриптор APPLET, который он обнаружит, в отдельном окне, в то время как веб-браузеры позволяют выполнять много аплетов на одной странице. До сих пор мы использовали только упрощенную форму дескриптора APPLET. Теперь пришло время рассмотреть его более внимательно.

Синтаксис полной формы дескриптора APPLET показан ниже. Элементы, взятые в квадратные скобки, являются необязательными.

```
<APPLET
    [CODEBASE = URL_базы_кода]
     CODE = файл_аплета
    [ALT = альтернативный_текст]
    [NAME = имя_экземпляра_аплета]
     WIDTH = пикселей HEIGHT = пикселей
    [ALIGN = выравнивание]
    [VSPACE = пикселей] [HSPACE = пикселей]
>
[<PARAM NAME = Имя_атрибута VALUE = Значение_атрибута>]
[<PARAM NAME = Имя_атрибута2 VALUE = Значение_атрибута>]
. . .
[Код HTML, отображаемый при отсутствии Java]
</APPLET>
```

Теперь рассмотрим каждую часть по очереди.

- CODEBASE — необязательный атрибут, задающий базовый URL кода аплета, который представляет собой каталог, где будет выполняться поиск исполняемого файла класса (указанного в дескрипторе CODE). Если этот атрибут не задан явно, в качестве атрибута CODEBASE используется каталог URL документа HTML. Атрибут CODEBASE не обязательно должен находиться на том же хосте, откуда прочитан документ HTML.
- CODE — обязательный атрибут, который задает имя файла, содержащего скомпилированный файл `.class` вашего аплета. Имя этого файла задается относительно базового URL кода аплета, являвшегося каталогом, где располагается файл HTML, либо каталогом, указанным в атрибуте CODEBASE.
- ALT — необязательный атрибут, используемый для указания краткого текстового сообщения, которое должно быть отображено, если браузер распознает дескриптор APPLET, но в данный момент не сможет выполнять аплеты Java. Это отличается от альтернативного кода HTML, который вы предоставляете для браузеров, вообще не поддерживающих аплеты.

- NAME — необязательный атрибут, используемый для указания имени экземпляра аплета. Аплеты должны именоваться таким образом, чтобы другие аплеты на той же странице могли находить их по именам и взаимодействовать с ними. Чтобы получить аплет по имени, используйте метод getApplet(), определенный в интерфейсе AppletContext.
- WIDTH и HEIGHT — обязательные атрибуты, задающие размер (в пикселях) отображаемой области аплета.
- ALIGN — необязательный атрибут, задающий выравнивание аплета. Этот атрибут трактуется точно так же, как в дескрипторе HTML IMG, и имеет следующие значения: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE и ABSBOTTOM.
- VSPACE и HSPACE — необязательные атрибуты. Атрибут VSPACE определяет пространство в пикселях над и под аплетом, а атрибут HSPACE — пространство в пикселях по бокам аплета. Они трактуются точно так же, как и атрибуты VSPACE и HSPACE дескриптора IMG.
- PARAM NAME и VALUE — дескриптор PARAM позволяет указать специфичные для аплета аргументы. Аплеты получают доступ к этим атрибутам при помощи метода getParametr().

Среди прочих допустимых атрибутов, еще можно упомянуть атрибут ARCHIVE, позволяющий задать один или более архивных файлов, и атрибут OBJECT, указывающий сохраненную версию аплета. В общем случае дескриптор APPLET должен включать только атрибут CODE или OBJECT, но не оба сразу.

Передача параметров аплетам

Как только что упоминалось, дескриптор HTML APPLET позволяет передавать параметры аплету. Для извлечения параметра служит метод getParameter(). Он возвращает значение указанного параметра в форме строки. Таким образом, для числовых и булевых значений вам придется преобразовывать их строковые представления во внутренние форматы. Рассмотрим пример передачи параметров.

```
// Использование параметров.
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/

public class ParamDemo extends Applet {
    String fontName;
    int fontSize;
    float leading;
    boolean active;

    // Инициализация строки для отображения.
    public void start() {
        String param;
```

```
        fontName = getParameter("fontName");
        if(fontName == null)
            fontName = "Not Found";
         // fontName = "Не найден";

        param = getParameter("fontSize");
        try {
            if(param != null)
                fontSize = Integer.parseInt(param);
            else
                fontSize = 0;
        } catch(NumberFormatException e) {
            fontSize = -1;
        }

        param = getParameter("leading");
        try {
            if(param != null)
                leading = Float.valueOf(param).floatValue();
            else
                leading = 0;
        } catch(NumberFormatException e) {
            leading = -1;
        }

        param = getParameter("accountEnabled");
        if(param != null)
            active = Boolean.valueOf(param).booleanValue();
    }

    // Отобразить параметры.
    public void paint(Graphics g) {
        g.drawString("Font name: " + fontName, 0, 10);
        g.drawString("Font size: " + fontSize, 0, 26);
        g.drawString("Leading: " + leading, 0, 42);
        g.drawString("Account Active: " + active, 0, 58);
     // g.drawString("Имя шрифта: " + fontName, 0, 10);
     // g.drawString("Размер шрифта: " + fontSize, 0, 26);
     // g.drawString("Отступ: " + leading, 0, 42);
     // g.drawString("Активный счет: " + active, 0, 58);
    }
}
```

Пример результирующего окна этого аплета показан на рис. 22.5.

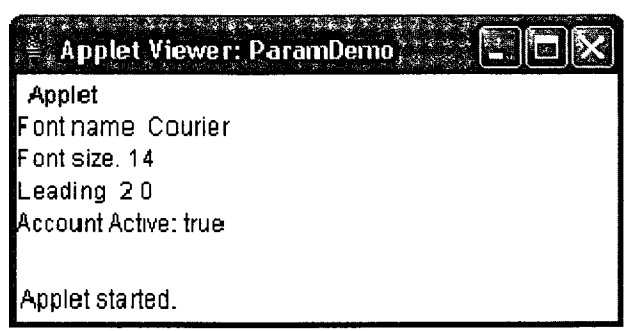


Рис. 22.5. Окно аплета, принимающего параметры

Как видно в приведенном выше коде, следует проверять возвращаемые значения метода `getParameter()`. Если параметр недоступен, метод `getParameter()` вернет значение `null`. Также должна быть предпринята попытка преобразования в числовые типы в операторе `try`, который перехватывает исключение `NumberFormatException`. Внутри аплетов никогда нельзя допускать необработанных исключений.

Усовершенствование аплета баннера

Параметр можно использовать для усовершенствования аплета баннера, показанного ранее. В предыдущей версии отображаемое сообщение было жестко закодировано в аплете. Передача сообщения в виде параметра позволяет аплету баннера отображать различные сообщения при каждом выполнении. Ниже приведена его усовершенствованная версия. Обратите внимание на то, что дескриптор `APPLET` в начале файла теперь определяет параметр по имени `message`, связанный со строкой в кавычках.

```
// Параметризованный баннер.
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamBanner" width=300 height=50>
<param name=message value="Java заставляет веб двигаться!">
</applet>
*/

public class ParamBanner extends Applet implements Runnable {
    String msg;
    Thread t = null;
    int state;
    volatile boolean stopFlag;

    // Установить цвета и инициализировать поток.
    public void init() {
        setBackground(Color.cyan);
        setForeground(Color.red);
    }

    // Запуск потока
    public void start() {
        msg = getParameter("message");
        if(msg == null) msg = "Message not found.";
        msg = " " + msg;
        t = new Thread(this);
        stopFlag = false;
        t.start();
    }

    // Точка входа для потока, запускающего баннер.
    public void run() {
        // Переотобразить баннер
        for( ; ; ) {
            try {
                repaint();
                Thread.sleep(250);
                if(stopFlag)
                    break;
            } catch(InterruptedException e) {}
```

```
        }
    }

    // Пауза в отображении баннера.
    public void stop() {
        stopFlag = true;
        t = null;
    }

    // Отобразить баннер.
    public void paint(Graphics g) {
        char ch;

        ch = msg.charAt(0);
        msg = msg.substring(1, msg.length());
        msg += ch;

        g.drawString(msg, 50, 30);
    }
}
```

Методы getDocumentBase() и getCodeBase()

Нередко приходится создавать аплеты, которые должны явно загружать медиаинформацию и текст. Java позволяет аплетам загружать данные из каталога, содержащего файл HTML, который запускает аплет (*база документа*), а также из каталога, из которого загружается класс аплета (*база кода*). Эти каталоги возвращаются как объекты класса `URL` (описаны в главе 20) из методов `getDocumentBase()` и `getCodeBase()`. Они могут быть связаны со строкой — именем файла, который вы хотите загрузить. Чтобы действительно загрузить другой файл, вы используете метод `showDocument()`, определенный в интерфейсе `AppletContext`, о котором поговорим в следующем разделе.

В приведенном ниже аплете иллюстрируется применение этих методов.

```
// Отображение баз документа и кода.
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="Bases" width=300 height=50>
</applet>
*/

public class Bases extends Applet {
    // Отображение базы документа и базы кода.
    public void paint(Graphics g) {
        String msg;

        URL url = getCodeBase(); // получить базу кода
        msg = "Code base: " + url.toString();
     // msg = "База кода: " + url.toString();
        g.drawString(msg, 10, 20);

        url = getDocumentBase(); // получить базу документа
        msg = "Document base: " + url.toString();
     // msg = "База документа: " + url.toString();
```

```
        g.drawString(msg, 10, 40);
    }
}
```

Примерный вывод этого аплета показан на рис. 22.6.

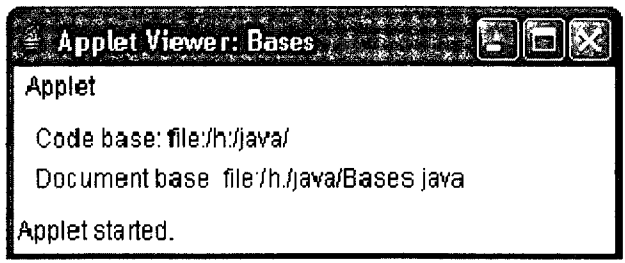


Рис. 22.6. Окно аплета, отображающего базы документа и кода

Интерфейс `AppletContext` и метод `showDocument()`

Одним из применений Java является использование активных изображений и анимации для обеспечения графических средств навигации в веб, которые более интересны, чем простые текстовые ссылки. Чтобы позволить вашему аплету передавать управление другому URL, следует использовать метод `showDocument()`, определенный в интерфейсе `AppletContext`. Интерфейс `AppletContext` позволяет получать информацию от исполняющей среды аплета. Методы, определенные в интерфейсе `AppletContext`, перечислены в табл. 22.2. Контекст текущего выполняемого аплета получается вызовом метода `getAppletContext()`, определенного в классе `Applet`.

Таблица 22.2. Методы, определенные интерфейсом `AppletContext`

Метод	Описание
`Applet getApplet(String` *имяАплета*`)`	Возвращает аплет, заданный именем *имяАплета*, если он находится внутри контекста текущего аплета. В противном случае возвращается значение `null`
`Enumeration<Applet> getApplets()`	Возвращает перечисление, содержащее все аплеты из контекста текущего аплета
`AudioClip getAudioClip(URL` *url*`)`	Возвращает объект интерфейса `AudioClip`, инкапсулирующий аудиоклип, находящийся в месте, указанном *url*
`Image getImage(URL` *url*`)`	Возвращает объект класса `Image`, инкапсулирующий графическое изображение, находящееся в месте, указанном *url*
`InputStream getStream(String` *ключ*`)`	Возвращает поток, связанный с *ключ*. Ключи привязываются к потокам с помощью метода `setStream()`. Если связанного с *ключ* потока не существует, возвращается значение `null`
`Iterator<String> getStreamKeys()`	Возвращает итератор для ключей, ассоциированных с вызывающим объектом. Ключи привязаны к потокам. См. также методы `getStream()` и `setStream()`

Окончание табл. 22.2

Метод	Описание
`void setStream(String` *ключ*`, InputStream` *поток*`) throws IOException`	Связывает поток, заданный параметром *поток*, с ключом, переданным параметром *ключ*. Ключ удаляется из вызывающего объекта, если *поток* содержит значение `null`
`void showDocument(URL` *url*`)`	Отображает в представлении документ, расположенный по адресу URL, переданному параметром *url*. Этот метод может не поддерживаться средствами просмотра аплетов
`void showDocument(URL` *url*`, String` *где*`)`	Отображает в представлении документ, расположенный по адресу URL, переданному параметром *url*. Этот метод может не поддерживаться средствами просмотра аплетов. Расположение документа указано параметром *где*, как описано в тексте
`void showStatus(String` *строка*`)`	Отображает содержимое параметра *строка* в окне (строке) состояния

Как только вы получаете контекст аплета, то сразу можете перенести в отображаемое представление другой документ, вызвав для этого метод `showDocument()`. Этот метод не имеет возвращаемого значения и не передает исключений в случае неудачи, поэтому используйте его осторожно. Доступны два метода `showDocument()`. Метод `showDocument(URL)` отображает документ, расположенный по указанному URL, а метод `showDocument(URL, String)` — документ, находящийся в указанном месте внутри окна браузера. Корректными аргументами параметра *где* являются `"_self"` (отобразить в текущем фрейме), `"_parent"` (отобразить в родительском фрейме), `"_top"` (отобразить во фрейме наивысшего уровня) и `"_blank"` (отобразить в новом окне браузера). Вы можете также задать имя, что позволит отобразить документ в новом окне браузера по его имени.

В следующем аплете демонстрируется применение интерфейса `AppletContext` и метода `showDocument()`. При выполнении он получает контекст текущего аплета и использует его для передачи управления файлу по имени `Test.html`. Этот файл должен располагаться в том же каталоге, что и аплет. Файл `Test.html` может содержать любой допустимый гипертекст.

```
/* Использование контекста аплета, getCodeBase()
и showDocument() для отображения файла HTML.
*/

import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="ACDemo" width=300 height=50>
</applet>
*/

public class ACDemo extends Applet {
    public void start() {
        AppletContext ac = getAppletContext();
        URL url = getCodeBase(); // получить url данного аплета

        try {
            ac.showDocument(new URL(url+"Test.html"));
        } catch(MalformedURLException e) {
            showStatus("URL not found"); // showStatus("URL не найден");
        }
    }
}
```

Интерфейс AudioClip

Интерфейс AudioClip определяет три метода: play() (воспроизведение клипа с начала), stop() (останов воспроизведения клипа) и loop() (непрерывное циклическое воспроизведение). После загрузки аудиоклипа с помощью метода getAudioClip(), эти методы можно применять для его воспроизведения.

Интерфейс AppletStub

Интерфейс AppletStub предоставляет средства, с помощью которых аплет и браузер (или средство просмотра аплетов) взаимодействуют между собой. Обычно ваш код не должен реализовывать этот интерфейс.

Консольный вывод

Хотя вывод в окно аплета должен осуществляться методами графического интерфейса пользователя, такими как drawString(), аплеты также могут использовать консольный вывод – в частности, для отладочных целей. Когда в аплете вызывается метод вроде System.out.println(), его вывод не посылается в окно аплета. Вместо этого вывод появляется либо в сеансе консоли, из которой вы запустили средство просмотра аплета, либо в консоли Java, которая доступна в некоторых браузерах. Использование консольного вывода в целях, отличных от отладки, не рекомендуется, поскольку он нарушает принципы дизайна графического интерфейса, к которым привыкли большинство пользователей.

ГЛАВА 23

Обработка событий

Настоящая глава посвящена важнейшему аспекту Java — событиям. Обработка событий является фундаментальной для всего программирования Java, поскольку это — основа аплетов и прочих типов программ с *графическим пользовательским интерфейсом* (Graphical User Interface — GUI). Как упоминалось в главе 22, аплеты представляют собой управляемые событиями программы, использующие графический интерфейс для взаимодействия с пользователем. Более того, любая программа с графическим пользовательским интерфейсом, такая как приложение Java, написанное для Windows, является управляемой событиями. Другими словами, вы не можете написать такую программу без четкого представления об обработке событий. События поддерживаются множеством пакетов, включая `java.util`, `java.awt` и `java.event`.

Большинство событий, на которые будет реагировать ваша программа, происходят при взаимодействии пользователя с программой на основе GUI. Именно события такого рода и рассматриваются в настоящей главе. События попадают в вашу программу множеством различных путей, каждый из которых зависит от конкретного события. Существует несколько типов событий, включая создаваемые мышью, клавиатурой и различными элементами управления GUI, такими как кнопка, полоса прокрутки или флажок.

Мы начнем эту главу с обзора механизма управления событиями Java. Затем рассмотрим основные классы и интерфейсы событий, используемые библиотекой AWT, и разработаем несколько примеров, демонстрирующих основы обработки событий. Эта глава также расскажет о том, как использовать классы адаптеров, вложенные классы и анонимные вложенные классы, чтобы упростить код обработки событий. Примеры, приведенные в остальной части книги, будут использовать эти приемы.

На заметку! Материал этой главы основан на событиях, имеющих отношение к программам на основе GUI. Но иногда события также используются в целях, не имеющих прямого отношения к программам подобного рода. Тем не менее во всех случаях применяются одни и те же приемы обработки событий.

Два механизма обработки событий

Прежде чем приступить к обсуждению обработки событий, сделаем одно важное замечание. Способ обработки событий существенно изменился от исходной версии Java (1.0) до более современных версий, начиная с 1.1. Метод обработки событий, принятый в версии 1.0, все еще поддерживается, хотя и не рекомендован для применения в новых программах. К тому же многие методы, поддерживающие старую модель событий 1.0, теперь объявлены устаревшими (deprecated). Современный подход состоит в том, что события должны обрабатываться всеми программами так, как описано в этой книге.

Модель делегирования событий

Современный подход к обработке событий основан на *модели* делегирования *событий* (delegation event model), определяющей стандартные и согласованные механизмы для создания и обработки событий. Его концепция проста: *источник* извещает о событии одного или несколько *слушателей* (listener). В этой схеме слушатель просто ожидает до тех пор, пока не получит извещение о событии. Как только извещение о событии получено, слушатель обрабатывает его и возвращает управление. Преимущество такого дизайна в том, что логика приложения, обрабатывающего события, четко отделена от логики пользовательского интерфейса, извещающего об этом событии. Элемент пользовательского интерфейса может "делегировать" обработку события отдельному фрагменту кода.

В модели делегирования событий слушатели должны регистрироваться источником для того, чтобы получать извещения о событиях. Это обеспечивает важное преимущество: уведомления посылаются только тем слушателям, которые желают получать их. Это более эффективный способ обработки событий, нежели тот, что был принят в старом подходе Java 1.0. Раньше извещение о событии распространялось по всей иерархии вложенности до тех пор, пока не было обработано компонентом. Это вынуждало все компоненты получать извещения о событии, которые они могли и не обрабатывать, что приводило к значительным затратам времени. Модель делегирования событий исключила подобную расточительность.

В последующих разделах определим события и опишем роли источников и слушателей.

На заметку! Java также позволяет обрабатывать события без применения модели делегирования. Это может быть сделано при помощи расширения компонента библиотеки AWT. Такая техника обсуждается в конце главы 25. Однако модель делегирования событий более предпочтительна по описанным выше причинам.

События

В модели делегирования *событие* — это объект, описывающий изменение состояния источника. Он может быть создан в результате взаимодействия пользователя с элементом графического интерфейса. К событиям приводят такие действия, как щелчок на экранной кнопке, ввод символа с клавиатуры, выбор элемента в списке и щелчок кнопкой мыши. Многие другие пользовательские операции также могут служить подобными примерами.

События могут также происходить и не в результате прямого взаимодействия с пользовательским интерфейсом. Например, событие может произойти по истечении времени таймера в результате превышения счетчиком некоторого значения, программного или аппаратного сбоя либо завершения некоторой операции. Вы можете определять собственные события, отвечающие специфике вашего приложения.

Источники событий

Источник (source) — это объект, извещающий о событии. Событие происходит при изменении внутреннего состояния объекта некоторым образом. Источники могут извещать о событиях нескольких типов.

Источник должен регистрировать слушателей, чтобы они получали извещение о событиях определенного рода. Каждый тип события имеет собственный метод регистрации. Вот его общая форма.

```
public void addТипListener(ТипListener el)
```

Здесь *Тип* — имя события, а `el` — ссылка на слушателя событий. Например, метод, регистрирующий слушателя событий клавиатуры, называется `addKeyListener()`, а метод, регистрирующий слушателя движения мыши, — `addMouseMotionListener()`. Когда событие происходит, все зарегистрированные слушатели получают копию объекта события. Это называется *групповой рассылкой* (multicasting) события. Во всех случаях уведомления отправляются только тем слушателям, которые зарегистрированы на их получение.

Некоторые источники допускают регистрацию только одного слушателя. Общая форма такого метода показана ниже.

```
public void addТипListener(ТипListener el)
    throws java.util.TooManyListenersException
```

Здесь *Тип* — это имя объекта события, а `el` — ссылка на слушателя события. Когда такое событие происходит, зарегистрированный слушатель получает уведомление. Это называется *индивидуальной рассылкой* (unicasting) события.

Источник должен также предоставлять метод, позволяющий слушателю отменить регистрацию для определенного типа событий. Общая форма этого метода такова.

```
public void removeТипListener(ТипListener el)
```

И снова *Тип* — это имя объекта события, а `el` — ссылка на слушателя события. Например, чтобы удалить слушателя клавиатуры, следует вызвать метод `removeKeyListener()`.

Метод, который добавляет или удаляет слушателей, предоставляется источником, извещающим о событии. Например, класс `Component` предлагает методы для добавления и удаления слушателей клавиатуры и мыши.

Слушатели событий

Слушатель (listener) — это объект, уведомляемый о событии. К нему предъявляются два основных требования. Во-первых, он должен быть зарегистрирован одним или более источниками событий, чтобы получать уведомления о событиях определенного рода. Во-вторых, он должен реализовать методы для получения и обработки таких уведомлений.

Методы, принимающие и обрабатывающие события, определены в наборе интерфейсов, находящихся в пакете `java.awt.event`. Например, интерфейс `MouseMotionListener` определяет два метода для получения уведомлений о перетаскивании объекта или перемещении мыши.

Любой объект может принимать и обрабатывать одно или оба эти события, если предоставляет реализацию этого интерфейса. В этой и последующих главах мы еще поговорим о многих других интерфейсах слушателей.

Классы событий

Классы, представляющие события, находятся в ядре механизма обработки событий Java. А потому дискуссия об обработке событий должна начинаться с классов событий. Важно понимать, однако, что Java определяет несколько типов событий и что не все классы событий будут упомянуты в настоящей главе. Наиболее широко используемыми событиями являются те, что определены библиотеками AWT и Swing. Здесь сосредоточимся на событиях библиотеки AWT. (Многие из них относятся также и к библиотеке Swing.) Несколько специфич-

ных для библиотеки Swing событий будут описаны в главе 30, когда речь пойдет о библиотеке Swing.

В корне иерархии классов событий Java лежит класс `EventObject`, расположенный в пакете `java.util`. Это – суперкласс для всех событий. Его единственный конструктор выглядит следующим образом.

```
EventObject(Object источник)
```

Параметр *источник* здесь представляет объект, извещающий о событии.

Класс `EventObject` содержит два метода: `getSource()` и `toString()`. Метод `getSource()` возвращает источник события. Его общая форма такова.

```
Object getSource()
```

Как можно ожидать, метод `toString()` возвращает строку – эквивалент события.

Класс `AWTEvent`, определенный в пакете `java.awt`, рассматривается в конце главы 25. А пока важно знать, что все классы, о которых пойдет речь в этом разделе, являются подклассами класса `AWTEvent`. Итак, подытожим.

- Класс `EventObject` – суперкласс для всех событий.
- Класс `AWTEvent` – суперкласс всех событий библиотеки AWT, обрабатываемых моделью делегирования событий.

Пакет `java.awt.event` определяет множество типов событий, происходящих во многих элементах пользовательского интерфейса. В табл. 23.1 показано несколько широко используемых классов событий и представлено краткое описание того, когда они происходят. Наиболее часто используемые конструкторы и методы каждого класса описаны в последующих разделах.

Таблица 23.1. Основные классы событий в пакете `java.awt.event`

Класс события	Описание
`ActionEvent`	Происходит при нажатии кнопки, двойном щелчке на элементе списка либо выборе пункта меню
`AdjustmentEvent`	Происходит при манипуляциях с полосой прокрутки
`ComponentEvent`	Происходит при сокрытии компонента, его перемещении, изменении его размера либо включении видимости
`ContainerEvent`	Происходит при добавлении или исключении компонента контейнера
`FocusEvent`	Происходит, когда компонент получает или теряет фокус клавиатурного ввода
`InputEvent`	Абстрактный суперкласс для всех классов ввода компонентов
`ItemEvent`	Происходит при щелчке на флажке или элементе списка, а также при выборе элемента списка и выборе или отмене выбора пункта меню
`KeyEvent`	Происходит при получении ввода с клавиатуры
`MouseEvent`	Происходит при перетаскивании, перемещении, щелчках, нажатии и отпускании кнопок мыши; также происходит, когда курсор мыши наводится на компонент либо покидает его
`MouseWheelEvent`	Происходит при прокрутке колесика мыши
`TextEvent`	Происходит при изменении значения текстовой области или текстового поля
`WindowEvent`	Происходит при активизации либо закрытии окна, а также при деактивизации, свертывании, развертывании окна или выходе из него

Класс ActionEvent

Объект класса ActionEvent создается при нажатии кнопки, двойном щелчке на элементе списка либо при выборе пункта меню. Класс ActionEvent определяет четыре целочисленные константы, которые могут быть использованы для идентификации любых модификаторов, ассоциированных с событием действия, – ALT_MASK, CTRL_MASK, META_MASK и SHIFT_MASK. Кроме того, имеется целочисленная константа ACTION_PERFORMED; которая может служить для идентификации событий действия.

Класс ActionEvent имеет три конструктора.

```
ActionEvent(Object источник, int тип, String команда)
ActionEvent(Object источник, int тип, String команда, int модификаторы)
ActionEvent(Object источник, int тип, String команда, long когда, int
модификаторы)
```

Здесь *источник* – это ссылка на объект, извещающий о событии. Тип события указан параметром *тип*, а его командная строка – параметром *команда*. Аргумент *модификаторы* указывает на нажатие модифицирующих клавиш (<Alt>, <Ctrl>, <Meta> и/или <Shift>) в момент события.

Вы можете получить имя команды для вызывающего объекта класса ActionEvent, используя метод getActionCommand(), показанный ниже.

```
String getActionCommand()
```

Например, при щелчке на кнопке происходит событие действия, которое имеет имя команды, соответствующее метке экранной кнопки.

Метод getModifiers() возвращает значение, указывающее на то, какие модифицирующие клавиши (<Alt>, <Ctrl>, <Meta> и/или <Shift>) были нажаты в момент события. Его форма такова.

```
int getModifiers()
```

Метод getWhen() возвращает время, когда произошло событие. Это называется *временной меткой* события. Метод getWhen() выглядит следующим образом.

```
long getWhen()
```

Класс AdjustmentEvent

Событие класса AdjustmentEvent передается полосой прокрутки. Существует пять типов событий настройки (adjustment). Класс AdjustmentEvent определяет целочисленные константы, которые могут использоваться для идентификации. Константы и их описания представлены в табл. 23.2.

Таблица 23.2. Константы, определенные классом AdjustmentEvent

Константа	Описание
BLOCK_DECREMENT	Пользователь щелкнул внутри полосы прокрутки для уменьшения ее значения
BLOCK_INCREMENT	Пользователь щелкнул внутри полосы прокрутки для увеличения ее значения
TRACK	Передвинут бегунок полосы
UNIT_DECREMENT	Был выполнен щелчок на кнопке в конце полосы для уменьшения ее значения
UNIT_INCREMENT	Был выполнен щелчок на кнопке в конце полосы для увеличения ее значения

Кроме того, имеется также целочисленная константа ADJUSTMENT_VALUE_CHANGED, которая указывает на то, что произошло изменение.

Вот единственный конструктор класса AdjustmentEvent.

```
AdjustmentEvent(Adjustable источник, int идентификатор, int тип, int данные)
```

Здесь *источник* — ссылка на объект, извещающий о событии. Параметр *идентификатор* определяет событие. Тип настройки указан параметром *тип*, а ассоциированные с ней данные — параметром *данные*.

Метод getAdjustable() возвращает объект, извещающий о событии. Его форма представлена ниже.

```
Adjustable getAdjustable()
```

Тип события настройки может быть получен методом getAdjustmentType(). Он возвращает одну из констант, определенных классом AdjustmentEvent. Его общая форма такова.

```
int getAdjustmentType()
```

Величина настройки может быть получена методом getValue(), показанным ниже.

```
int getValue()
```

Например, когда выполняются манипуляции с полосой прокрутки, этот метод возвращает значение, представленное изменением.

Класс ComponentEvent

Объект класса ComponentEvent создается при изменении размера, положения или видимости компонента. Существует четыре типа событий компонентов. Для их идентификации класс ComponentEvent определяет целочисленные константы. Константы и их описания представлены в табл. 23.3.

Таблица 23.3. Константы, определенные классом ComponentEvent

Константа	Описание
COMPONENT_HIDDEN	Компонент скрыт
COMPONENT_MOVED	Компонент перемещен
COMPONENT_RESIZED	Изменен размер компонента
COMPONENT_SHOWN	Компонент стал видимым

Класс ComponentEvent имеет следующий конструктор.

```
ComponentEvent(Component источник, int тип)
```

Здесь *источник* — ссылка на объект, извещающий о событии. Тип события указан параметром *тип*.

Класс ComponentEvent — это прямой или непрямой суперкласс для классов ContainerEvent, FocusEvent, KeyEvent, MouseEvent, WindowEvent и др.

Метод getComponent() возвращает компонент, извещающий о событии. Выглядит он так.

```
Component getComponent()
```

Класс ContainerEvent

Объект класса ContainerEvent создается при добавлении компонента в контейнер или его удалении оттуда. Существует два типа событий контейнера. Класс

ContainerEvent определяет целочисленные константы, которые могут использоваться для его идентификации, — COMPONENT_ADDED и COMPONENT_REMOVED. Они определяют, добавлен ли компонент к контейнеру или же исключен из него.

Класс ContainerEvent — это подкласс класса ComponentEvent, имеющий следующий конструктор.

```
ContainerEvent(Component источник, int тип, Component компаратор)
```

Здесь *источник* — ссылка на контейнер, который известил о событии. Тип события указан параметром *тип*, а компонент, добавляемый в контейнер либо удаляемый из него, — параметром *компаратор*.

Вы можете получить ссылку на контейнер, создавший это событие, из метода getContainer(), показанного ниже.

```
Container getContainer()
```

Метод getChild() возвращает ссылку на компонент, который был добавлен или удален из контейнера. Его общая форма такова.

```
Component getChild()
```

Класс FocusEvent

Объект класса FocusEvent создается при получении или потере компонентом фокуса. Эти события определяются целочисленными константами FOCUS_GAINED и FOCUS_LOST.

Класс FocusEvent — это подкласс класса ComponentEvent, имеющий следующие конструкторы.

```
FocusEvent(Component источник, int тип)
FocusEvent(Component источник, int тип, boolean временныйФлаг)
FocusEvent(Component источник, int тип, boolean временныйФлаг, Component
другое)
```

Здесь *источник* — это ссылка на компонент, извещающий о событии. Тип события указан параметром *тип*. Аргумент *временныйФлаг* установлен в состояние true, если событие фокуса является временным. В противном случае он устанавливается в состояние false. (Событие временного фокуса происходит в результате другой операции пользовательского интерфейса. Например, предположим, что фокус находится на текстовом поле. Если пользователь перемещает мышь, чтобы подвинуть полосу прокрутки, то фокус временно теряется.)

Другой компонент, участвующий в изменении фокуса и называемый противоположным компонентом, передается параметром *другое*. Таким образом, если происходит событие FOCUS_GAINED, то параметр *другое* будет ссылаться на компонент, теряющий фокус. В противоположность этому, если происходит событие FOCUS_LOST, то параметр *другое* ссылается на компонент, получающий фокус.

Вы можете определить другие компоненты вызовом метода getOppositeComponent(), показанного ниже.

```
Component getOppositeComponent()
```

Метод возвращает противоположный компонент.

Метод isTemporary() указывает на то, является ли изменение фокуса временным. Его форма такова.

```
boolean isTemporary()
```

Метод возвращает значение true, если изменение временно. В противном случае он возвращает значение false.

Класс InputEvent

Абстрактный класс InputEvent является подклассом класса ComponentEvent и суперклассом для событий ввода компонента. Его подклассы — KeyEvent и MouseEvent.

Класс InputEvent определяет несколько строковых констант, представляющих любые модификаторы, такие как признак нажатия управляющих клавиш, которые могут быть ассоциированы с событием. Изначально класс InputEvent определяет следующие восемь значений для представления модификаторов.

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

Однако поскольку существует возможность конфликтов в используемых модификаторах между клавиатурными событиями, событиями мыши, а также другие проблемы, 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 рекомендуется использовать новые расширенные модификаторы вместо исходных.

Чтобы проверить, какая клавиша модификатора была нажата в момент события, используйте методы isAltDown(), isAltGraphDown(), isControlDown(), isMetaDown() и isShiftDown(). Формы этих методов показаны ниже.

```
boolean isAltDown()
boolean isAltGraphDown()
boolean isControlDown()
boolean isMetaDown()
boolean isShiftDown()
```

Вы можете получить значение, содержащее все флаги исходных модификаторов, вызовом метода getModifiers(). Он выглядит так.

```
int getModifiers()
```

Расширенные модификаторы можно получить методом getModifiersEx(), показанным ниже.

```
int getModifiersEx()
```

Класс ItemEvent

Объект класса ItemEvent создается при щелчке на флажке или элементе списка либо при выборе отмечаемого пункта меню. (Флажки и окна списков будут описаны далее.) Существует два типа событий элементов, которые идентифицируются целочисленными константами, описанными в табл. 23.4.

Таблица 23.4. Константы, определенные классом ItemEvent

Константа	Описание
DESELECTED	Пользователь отменил выбор элемента
SELECTED	Пользователь выбрал элемент

Кроме того, класс ItemEvent определяет одну целочисленную константу ITEM_STATE_CHANGED, которая сигнализирует об изменении состояния.

Класс ItemEvent имеет следующий конструктор.

```
ItemEvent(ItemSelectable источник, int тип, Object элемент, int состояние)
```

Здесь *источник* — ссылка на компонент, известивший о событии. Например, это может быть список или элемент выбора. Тип события указывается параметром *тип*. Конкретный элемент, приведший к событию, передается в параметре *элемент*. Текущее состояние элемента содержится в параметре *состояние*.

Метод getItem() может быть использован для получения ссылки на элемент, приведший к событию. Его сигнатура выглядит так.

```
Object getItem()
```

Метод getItemSelectable() может быть использован для получения ссылки на объект ItemSelectable, известивший о событии. Его общая форма показана ниже.

```
ItemSelectable getItemSelectable()
```

Списки и флажки являются примерами элементов пользовательского интерфейса, реализующих интерфейс ItemSelectable.

Метод getStateChanged() возвращает измененное состояние (т.е. значение SELECTED или DESELECTED) для события. Выглядит он так.

```
int getStateChanged()
```

Класс KeyEvent

Объект класса KeyEvent создается при клавиатурном вводе. Существует три типа клавиатурных событий, идентифицируемых целочисленными константами, — KEY_PRESSED, KEY_RELEASED и KEY_TYPED. События первых двух типов происходят при нажатии и отпускании клавиши. Последний же тип происходит при вводе символа. Помните, что нажатие не всех клавиш приводит к вводу символа. Например, нажатие клавиши <Shift> не вводит никакого символа.

Класс KeyEvent определяет множество других целочисленных констант. Например, константы VK_0–VK_9 и VK_A–VK_Z определяют эквиваленты ASCII чисел и букв. А вот еще несколько других констант.

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 задают *коды виртуальных клавиш*, не зависящие от любых модификаторов вроде <Control>, <Alt> или <Shift>.

Класс KeyEvent является подклассом класса InputEvent. Вот один из его конструкторов.

```
KeyEvent(Component источник, int тип, long когда, int модификаторы, int
код, char символ)
```

Здесь *источник* — ссылка на компонент, извещающий о событии. Тип события указывается параметром *тип*. Системное время, когда была нажата клавиша, передается в параметре *когда*. Аргумент *модификаторы* указывает, какие модификаторы были нажаты в момент события клавиши. Код виртуальной клавиши, такой как VK_UP, VK_A и так далее, передается в параметре *код*. Символьный эквивалент (если есть) передается в параметре *символ*. Если никакого корректного символа событие не содержит, то *символ* содержит значение CHAR_UNDEFINED. Для событий KEY_TYPED параметр *код* будет содержать значение VK_UNDEFINED.

Класс KeyEvent определяет несколько методов, но, вероятно, наиболее часто используемые из них — это метод getKeyChar(), возвращающий символ клавиши, и метод getKeyCode(), возвращающий код клавиши. Их общая форма представлена ниже.

```
char getKeyChar()
int getKeyCode()
```

Если никаких корректных символов нажатие клавиши не создает, то метод getKeyChar() возвращает значение CHAR_UNDEFINED. При событии KEY_TYPED метод getKeyCode() возвращает значение VK_UNDEFINED.

Класс MouseEvent

Существует восемь типов событий мыши. Класс MouseEvent определяет для их идентификации набор целочисленных констант, которые описаны в табл. 23.5.

Таблица 23.5. Константы, определенные классом MouseEvent

Константа	Описание
MOUSE_CLICKED	Пользователь щелкнул кнопкой мыши
MOUSE_DRAGGED	Пользователь перетащил мышь при нажатой кнопке
MOUSE_ENTERED	Курсор мыши вошел в компонент
MOUSE_EXITED	Курсор мыши покинул компонент
MOUSE_MOVED	Мышь перемещена
MOUSE_PRESSED	Кнопка мыши нажата
MOUSE_RELEASED	Кнопка мыши отпущена
MOUSE_WHEEL	Выполнена прокрутка колесика мыши

Класс MouseEvent — это подкласс класса InputEvent. Вот один из его конструкторов.

```
MouseEvent(Component источник, int тип, long когда, int модификаторы,
int x, int y, int щелчки, boolean вызовМеню)
```

Здесь *источник* — ссылка на компонент, извещающий о событии. Тип события указан параметром *тип*. Системное время, когда произошло событие, передается в параметре *когда*. Аргумент *модификаторы* указывает, какие клавиши модификаторов были нажаты в момент события мыши. Координаты курсора мыши передаются параметрами *x* и *y*. Счетчик щелчков передается в параметре *щелчки*. Флаг *вызовМеню* указывает, должно ли данное событие вызывать всплывающее меню на данной платформе.

Два часто используемых метода этого класса — это методы getX() и getY(). Они возвращают координаты X и Y курсора мыши на момент события. Их форма такова.

```
int getX()
int getY()
```

В качестве альтернативы для получения координат курсора мыши вы можете использовать метод getPoint(). Он показан ниже.

```
Point getPoint()
```

Этот метод возвращает объект класса Point, содержащий координаты X, Y в своих целочисленных членах x и y.

Метод `translatePoint()` изменяет местоположение события. Его форма показана ниже.

```
void translatePoint(int x, int y)
```

Здесь аргументы *x* и *y* добавляются к первоначальным координатам события.

Метод `getClickCount()` получает количество щелчков мыши для данного события. Его сигнатура показана ниже.

```
int getClickCount()
```

Метод `isPopupTrigger()` проверяет, вызывает ли данное событие всплывающее меню на текущей платформе. Его форма такова.

```
boolean isPopupTrigger()
```

Также доступен метод `getButton()`, показанный ниже.

```
int getButton()
```

Он возвращает значение, представляющее кнопку, вызвавшую событие. Возвращаемое значение будет одной из констант, определенных в классе `MouseEvent`.

NOBUTTON	BUTTON1	BUTTON2	BUTTON3

Значение `NOBUTTON` указывает на то, что никакая кнопка не была нажата или отпущена. Доступны также три метода класса `MouseEvent`, которые получают координаты мыши относительно экрана, а не компонента. Они показаны ниже.

```
Point getLocationOnScreen()
int getXOnScreen()
int getYOnScreen()
```

Метод `getLocationOnScreen()` возвращает объект класса `Point`, содержащий обе координаты – X и Y. Другие два метода возвращают по одной координате соответственно.

Класс `MouseWheelEvent`

Этот класс инкапсулирует событие колесика мыши. Он является подклассом класса `MouseEvent`. Не все мыши оснащены колесиками. Если у мыши есть колесико, оно располагается между левой и правой кнопками. Эти колесики используются для прокрутки (изображения, текста, таблиц и т.п.). Класс `MouseWheelEvent` определяет две целочисленные константы, описанные в табл. 23.6.

Таблица 23.6. Константы, определенные классом `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

Вот один из конструкторов, определенных в классе `MouseWheelEvent`.

```
MouseWheelEvent(Component источник, int тип, long когда,
                int модификаторы, int x, int y, int щелчки,
                boolean вызовМеню, int какПрокрутить,
                int сколько, int количество)
```

Здесь *источник* – ссылка на компонент, извещающий о событии. Тип события указан параметром *тип*. Системное время события передается в параметре *когда*. Аргумент *модификаторы* указывает, какие клавиши модификаторов были нажаты в момент события мыши. Координаты курсора мыши передаются параметрами *x* и *y*.

Счетчик щелчков передается в параметре *щелчки*. Флаг *вызовМеню* указывает на то, должно ли данное событие вызывать всплывающее меню на текущей платформе. Значением параметра *какПрокрутить* должно быть либо WHEEL_UNIT_SCROLL, либо WHEEL_BLOCK_SCROLL. Количество единиц прокрутки передается в параметре *сколько*. Параметр *количество* указывает количество единиц прокрутки, на которое повернуто колесико.

Класс MouseWheelEvent определяет методы, предоставляющие доступ к событию колесика. Чтобы получить количество единиц прокрутки, вызовите метод getWheelRotation(), показанный ниже.

```
int getWheelRotation()
```

Он возвращает количество единиц прокрутки. Если значение положительно, значит, колесико повернуто против часовой стрелки, если же отрицательно — то по часовой стрелке. В комплекте JDK 7 представлен метод getPreciseWheelRotation(), поддерживающий колесико с высокой разрешающей способностью. Он работает, как и метод getWheelRotation(), но возвращает значение типа double.

Чтобы получить тип прокрутки, вызовите метод getScrollType().

```
int getScrollType()
```

Он возвращает либо значение WHEEL_UNIT_SCROLL, либо значение WHEEL_BLOCK_SCROLL. Если типом прокрутки является WHEEL_UNIT_SCROLL, то вы получите количество единиц прокрутки непосредственно методом getScrollAmount(). Этот метод выглядит следующим образом.

```
int getScrollAmount()
```

Класс TextEvent

Экземпляры этого класса описывают текстовые события. Они создаются текстовыми полями и областями, когда в них осуществляется ввод пользователем или программой. Класс TextEvent определяет целочисленную константу TEXT_VALUE_CHANGED.

Единственный конструктор этого класса показан ниже.

```
TextEvent(Object источник, int тип)
```

Здесь *источник* — ссылка на объект, извещающий о событии. Тип события указывается параметром *тип*.

Объект класса TextEvent не включает символов, находящихся в данный момент в текстовом компоненте, известившем о событии. Вместо этого ваша программа должна использовать другие методы, ассоциированные с текстовым компонентом, для извлечения информации. Эта операция отличается от других объектов событий, о которых речь пойдет в следующем разделе. По этой причине никакие методы класса TextEvent мы пока не обсуждаем. Уведомления о текстовом событии следует считать сигналом слушателю о том, что последний должен извлечь информацию из указанного текстового компонента.

Класс WindowEvent

Существует десять типов событий окон. Класс WindowEvent определяет целочисленные константы, которые могут использоваться для их идентификации. Эти константы вместе с их описаниями перечислены в табл. 23.7.

Таблица 23.7. Константы, определенные классом `WindowEvent`

Константа	Описание
`WINDOW_ACTIVATED`	Окно было активизировано
`WINDOW_CLOSED`	Окно было закрыто
`WINDOW_CLOSING`	Пользователь запросил закрытие окна
`WINDOW_DEACTIVATED`	Окно деактивизировано
`WINDOW_DEICONIFIED`	Окно развернуто
`WINDOW_GAINED_FOCUS`	Окно получило фокус ввода
`WINDOW_ICONIFIED`	Окно свернуто
`WINDOW_LOST_FOCUS`	Окно утратило фокус ввода
`WINDOW_OPENED`	Окно было открыто
`WINDOW_STATE_CHANGED`	Состояние окна изменилось

Класс `WindowEvent` — это подкласс класса `ComponentEvent`. Он определяет несколько конструкторов. Рассмотрим первый.

```
WindowEvent(Window источник, int тип)
```

Здесь *источник* — ссылка на объект, известивший о событии. Тип события передается в параметре *тип*. Следующие три конструктора обеспечивают более подробный контроль.

```
WindowEvent(Window источник, int тип, Window другое)
WindowEvent(Window источник, int тип, int изСостояния, int вСостояние)
WindowEvent(Window источник, int тип, Window другое, int изСостояния,
int вСостояние)
```

Здесь параметр *другое* определяет противоположное окно при событии фокуса или активизации. Параметр *изСостояния* определяет предыдущее состояние окна, а параметр *вСостояние* — новое состояние, которое окно получает при смене состояния.

Часто используемый метод этого класса — `getWindow()`. Он возвращает объект класса `Window`, известивший о событии. Вот его общая форма.

```
Window getWindow()
```

Класс `WindowEvent` также определяет методы, возвращающие противоположное окно (при событиях фокуса или активизации), предыдущее состояние окна, а также его текущее состояние. Эти методы показаны ниже.

```
Window getOppositeWindow()
int getOldState()
int getNewState()
```

Источники событий

В табл. 23.8 перечислены некоторые компоненты пользовательского интерфейса, способные извещать о событиях, описанных в предыдущем разделе. В дополнение к элементам графического пользовательского интерфейса, любой класс, унаследованный от класса `Component`, такой как класс `Applet`, способен извещать о событиях. Например, вы можете получать события клавиатуры и мыши от аплета. (Также вы можете создавать собственные компоненты, которые извещают о событиях.) В этой главе поговорим только о событиях клавиатуры и мыши, а в следующих главах речь пойдет об обработке событий от источников, перечисленных в табл. 23.8.

Таблица 23.8. Примеры источников событий

Источник событий	Описание
Кнопка	Извещает о событии действия при нажатии кнопки
Флажок	Извещает о событиях элемента при установке и сбросе флажка
Переключатель	Извещает о событиях элемента при изменении выбора
Список	Извещает о событиях действия при двойном щелчке на элементе; извещает о событии элемента при выделении или снятии выделения с элемента
Пункт меню	Извещает о событиях действия при выборе пункта меню; извещает о событии элемента при установке и сбросе флажка с пункта меню
Полоса прокрутки	Извещает о событиях настройки при манипуляциях с полосой прокрутки
Текстовые компоненты	Извещает о текстовых событиях, когда пользователь вводит символ
Окно	Извещает о событиях окна при активизации, закрытии, деактивизации, развертывании, свертывании, открытии окна или выходе из окна

Интерфейсы слушателей событий

Как уже объяснялось, модель делегирования событий состоит из двух частей: источников и слушателей. Слушатели создаются при реализации одного или нескольких интерфейсов, определенных в пакете `java.awt.event`. Когда происходит событие, его источник вызывает соответствующий метод, определенный в слушателе, и передает объект события в качестве аргумента. В табл. 23.9 перечислены часто используемые интерфейсы слушателей и представлено краткое описание методов, определяемых ими. В следующих разделах рассматриваются специфические методы, содержащиеся в каждом интерфейсе.

Таблица 23.9. Часто используемые интерфейсы слушателей событий

Интерфейс	Описание
`ActionListener`	Определяет один метод для принятия событий действия
`AdjustmentListener`	Определяет один метод для принятия событий настройки
`ComponentListener`	Определяет четыре метода для распознавания сокрытия, перемещения, изменения размера или отображения компонента
`ContainerListener`	Определяет два метода для распознавания того, когда компонент добавляется в контейнер либо исключается из него
`FocusListener`	Определяет два метода для распознавания получения или утраты компонентом фокуса клавиатурного ввода
`ItemListener`	Определяет один метод, указывающий момент изменения состояния элемента
`KeyListener`	Определяет три метода, распознающих нажатие, отпускание клавиши либо ввод символа

Окончание табл. 23.9

Интерфейс	Описание
`MouseListener`	Определяет пять методов, распознающих щелчок мыши, момент входа курсора мыши на поле компонента, его выхода за пределы компонента, нажатия и отпускания кнопок мыши
`MouseMotionListener`	Определяет два метода для распознавания перетаскивания или движения мыши
`MouseWheelListener`	Определяет один метод, распознающий движение колесика мыши
`TextListener`	Определяет один метод, распознающий изменение текстового значения
`WindowFocusListener`	Определяет два метода для распознавания получения или утраты окном фокуса ввода
`WindowListener`	Определяет семь методов, распознающих активизацию окна, закрытие, деактивизацию, развертывание, свертывание, открытие или выход

Интерфейс `ActionListener`

Этот интерфейс определяет метод `actionPerformed()`, вызываемый при событии действия. Его общая форма показана ниже.

```
void actionPerformed(ActionEvent ae)
```

Интерфейс `AdjustmentListener`

Этот интерфейс определяет метод `adjustmentValueChanged()`, вызываемый при событии настройки. Его общая форма такова.

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

Интерфейс `ComponentListener`

Этот интерфейс определяет четыре метода, вызываемых при изменении размера компонента, его перемещении, отображении или сокрытии. Их общая форма представлена ниже.

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

Интерфейс `ContainerListener`

Этот интерфейс содержит два метода. Когда компонент добавляется к контейнеру, вызывается метод `componentAdded()`. Когда же компонент удаляется из контейнера, вызывается метод `componentRemoved()`. Их общие формы выглядят следующим образом.

```
void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)
```

Интерфейс FocusListener

Этот интерфейс определяет два метода. Когда компонент получает фокус клавиатурного ввода, вызывается метод focusGained(). Когда компонент теряет фокус ввода, вызывается метод focusLost(). Обобщенная форма этих методов показана ниже.

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```

Интерфейс ItemListener

Этот интерфейс определяет метод itemStateChanged(), вызываемый при изменении состояния элемента. Его общая форма такова.

```
void itemStateChanged(ItemEvent ie)
```

Интерфейс KeyListener

Этот интерфейс определяет три метода. Методы keyPressed() и keyReleased() вызываются, соответственно, при нажатии и отпускании клавиш. Метод keyTyped() вызывается при вводе символа.

Например, если пользователь нажимает и отпускает клавишу <A>, последовательно происходит три события: нажатие клавиши, ввод символа, отпускание клавиши. Если пользователь нажимает и отпускает клавишу <Home>, происходит последовательно только два события: нажатие клавиши и ее отпускание.

Общие формы этих методов выглядят следующим образом.

```
void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)
```

Интерфейс MouseListener

Этот интерфейс определяет пять методов. Если кнопка мыши нажата и отпущена в одной и той же точке, вызывается метод mouseClicked(). Когда курсор мыши входит на поле компонента, вызывается метод mouseEntered(). Когда же он покидает поле компонента, вызывается метод mouseExited(). Методы mousePressed() и mouseReleased() вызываются, соответственно, когда кнопка мыши нажимается и отпускается. Общие формы этих методов представлены ниже.

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

Интерфейс MouseMotionListener

Этот интерфейс определяет два метода. Метод mouseDragged() вызывается множество раз при перетаскивании объекта мышью. Метод mouseMoved() вызывается множество раз при перемещении мыши. Их общая форма такова.

```
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

Интерфейс MouseWheelListener

Этот интерфейс определяет метод mouseWheelMoved(), вызываемый при прокрутке колесика мыши. Его общая форма показана ниже.

```
void mouseWheelMoved(MouseWheelEvent mwe)
```

Интерфейс TextListener

Этот интерфейс определяет метод textChanged(), вызываемый при изменении содержимого текстовой области или текстового поля. Его общая форма выглядит следующим образом.

```
void textChanged(TextEvent te)
```

Интерфейс WindowFocusListener

Этот интерфейс определяет два метода — windowGainedFocus() и windowLostFocus(). Они вызываются, когда окно получает и утрачивает фокус ввода. Их общая форма показана ниже.

```
void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
```

Интерфейс WindowListener

Этот интерфейс определяет семь методов. Методы windowActivated() и windowDeactivated() вызываются, когда окно, соответственно, активизируется и деактивизируется.

Если окно сворачивается в пиктограмму, вызывается метод windowIconified(). Когда окно разворачивается, вызывается метод windowDeIconified(). Когда окно открывается и закрывается, вызываются методы windowOpened() и windowClosed(). Метод windowClosing() вызывается при закрытии окна. Общие формы этих 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 событий

Теперь, когда вы ознакомились с теорией, лежащей в основе модели делегирования событий, и получили представление о различных ее компонентах, наступило время обратиться к практике. Использовать модель делегирования событий довольно легко. Нужно просто выполнить следующие два действия.

1. Реализовать соответствующий интерфейс в слушателе, чтобы он мог принимать события нужного типа.
2. Реализовать код регистрации и отмены регистрации (при необходимости) слушателя как получателя уведомлений о событии.

Следует помнить, что источник может извещать о нескольких типах событий. Каждое событие должно быть зарегистрировано отдельно. К тому же объект может подписаться на получение нескольких типов событий и должен реализовать все интерфейсы, необходимые для получения этих событий.

Чтобы увидеть, как модель делегирования событий работает на практике, мы разберем примеры, обрабатывающие два часто используемых генератора событий — событий мыши и клавиатуры.

Обработка событий мыши

Чтобы обработать события мыши, следует реализовать интерфейсы `MouseListener` и `MouseMotionListener`. (Вы можете также реализовать интерфейс `MouseWheelListener`, но мы не станем делать этого здесь.) Следующий аплет демонстрирует весь процесс. Он отображает текущие координаты мыши в строке состояния. Всякий раз, когда нажимается кнопка, отображается слово `"Down"` в точке, где находится курсор мыши. При каждом отпускании кнопки отображается слово `"Up"`. При щелчке на кнопке, в левом верхнем углу отображаемой области аплета выводится сообщение `"Mouse clicked"`.

При входе и выходе курсора мыши из поля окна аплета в левом верхнем углу отображаемой области аплета также выводится соответствующее сообщение. При перетаскивании мышью отображается символ `"*"`, сопровождающий курсор мыши. Обратите внимание: две переменные — `mouseX` и `mouseY` — сохраняют местоположение курсора мыши, когда происходят события нажатия и отпускания кнопки, а также события перетаскивания. Эти координаты затем используются методом `paint()` для отображения вывода в точке возникновения события.

```
// Использование обработчиков событий мыши.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/

public class MouseEvents extends Applet
implements MouseListener, MouseMotionListener {

    String msg = "";
    int mouseX = 0, mouseY = 0; // координаты курсора мыши

    public void init() {
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    // Обработка щелчка мыши.
    public void mouseClicked(MouseEvent me) {
        // сохранить координаты
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse clicked.";
     // msg = "Щелчок кнопкой мыши.";
        repaint();
    }

    // Обработка входа курсора мыши.
    public void mouseEntered(MouseEvent me) {
```

```
    // сохранить координаты
    mouseX = 0;
    mouseY = 10;
    msg = "Mouse entered.";
 // msg = "Курсор мыши вошел.";
    repaint();
}

// Обработка выхода курсора мыши.
public void mouseExited(MouseEvent me) {
    // сохранить координаты
    mouseX = 0;
    mouseY = 10;
    msg = "Mouse exited.";
 // msg = "Курсор мыши вышел.";
    repaint();
}

// Обработка нажатия кнопки.
public void mousePressed(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
 // msg = "Вниз";
    repaint();
}

// Обработка отпускания кнопки.
public void mouseReleased(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up";
 // msg = "Вверх";
    repaint();
}

// Обработка перетаскивания мышью.
public void mouseDragged(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "*";
    showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
 // showStatus("Перетаскивание мыши в " + mouseX + ", " + mouseY);
    repaint();
}

// Обработка движения мыши.
public void mouseMoved(MouseEvent me) {
    // Показать состояние
    showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
 // showStatus("Перемещение мыши в " + me.getX() + ", " + me.getY());
}

// Отобразить msg в окне аплета в текущей позиции X,Y.
public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
}
```

Пример работы этого аплета показан на рис. 23.1.

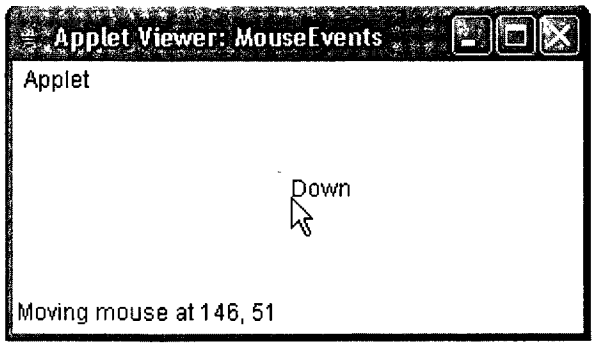


Рис. 23.1. Пример работы аплета, использующего обработчики событий мыши

Рассмотрим этот пример внимательнее. Класс `MouseEvents` расширяет класс `Applet` и реализует интерфейсы `MouseListener` и `MouseMotionListener`. Эти два интерфейса содержат методы, принимающие и обрабатывающие различные типы событий мыши. Обратите внимание на то, что аплет одновременно является и источником, и слушателем этих событий. Это работает, потому что класс `Component`, применяющий методы `addMouseListener()` и `addMouseMotionListener()`, является суперклассом класса `Applet`. Использование одного и того же объекта в качестве источника и слушателя событий типично для аплетов.

Внутри метода `init()` аплет регистрирует самого себя в качестве слушателя событий мыши. Это осуществляется методами `addMouseListener()` и `addMouseMotionListener()`, которые, как уже упоминалось, являются членами класса `Component`. Эти методы показаны ниже.

```
void addMouseListener(MouseListener ml)
void addMouseMotionListener(MouseMotionListener mml)
```

Здесь *ml* — ссылка на объект, принимающий события мыши, а *mml* — ссылка на объект, принимающий события перемещения мыши. В данной программе в обоих методах используется один и тот же объект.

Затем аплет реализует все методы, определенные в интерфейсах `MouseListener` и `MouseMotionListener`. Таким образом, существуют обработчики для всех событий мыши. Каждый из них обрабатывает собственное событие, а затем возвращает управление.

Обработка событий клавиатуры

Для обработки событий клавиатуры, используется та же общая архитектура, что и в примере с событиями мыши, приведенном в предыдущем разделе. Отличие, конечно, в том, что здесь вы будете реализовывать интерфейс `KeyListener`.

Прежде чем обращаться к примеру, полезно еще раз рассмотреть процесс извещения о клавиатурных событиях. При нажатии клавиши происходит событие `KEY_PRESSED`. Это приводит к вызову обработчика событий `keyPressed()`. При отпускании клавиши происходит событие `KEY_RELEASED` и выполняется обработчик `keyReleased()`. Если нажатие клавиши создает символ, то также происходит событие `KEY_TYPED` и выполняется обработчик `keyTyped()`. Таким образом, всякий раз, когда пользователь нажимает клавишу, происходит как минимум два, а то и три события. Если вас интересуют действительные символы, введенные с клавиатуры, то можете игнорировать информацию о нажатии и отпускании клавиш. Но если ваша программа должна обрабатывать специальные клавиши вроде клавиш

со стрелками или функциональных клавиш, то их следует отслеживать в обработчике keyPressed().

В следующем аплете демонстрируется клавиатурный ввод. Он отображает нажатые клавиши в окне аплета и демонстрирует в строке состояния, нажата клавиша или отпущена.

```
// Использование обработчиков событий клавиатуры.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/

public class SimpleKey extends Applet
implements KeyListener {

    String msg = "";
    int X = 10, Y = 20; // координаты вывода

    public void init() {
        addKeyListener(this);
    }

    public void keyPressed(KeyEvent ke) {
        showStatus("Key Down");
     // showStatus("Клавиша нажата");
    }

    public void keyReleased(KeyEvent ke) {
        showStatus("Key Up");
     // showStatus("Клавиша отпущена");
    }

    public void keyTyped(KeyEvent ke) {
        msg += ke.getKeyChar();
        repaint();
    }
    // Отображение нажатых клавиш.
    public void paint(Graphics g) {
        g.drawString(msg, X, Y);
    }
}
```

Пример работы этого аплета показан на рис. 23.2.

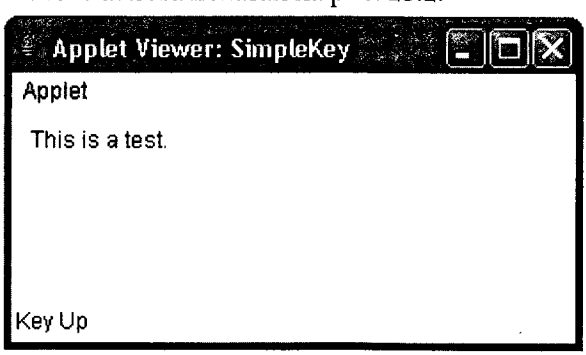


Рис. 23.2. Пример работы аплета, использующего обработчики событий клавиатуры

Если хотите обрабатывать специальные клавиши, такие как клавиши со стрелками и функциональные клавиши, то следует реагировать на них в обработчике `keyPressed()`. Такие клавиши в обработчике `keyTyped()` не доступны. Чтобы идентифицировать эти клавиши, можно использовать коды виртуальных клавиш. Например, следующий аплет выводит имена некоторых специальных клавиш.

```
// Использование некоторых кодов виртуальных клавиш.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="KeyEvents" width=300 height=100>
</applet>
*/

public class KeyEvents extends Applet
implements KeyListener {

    String msg = "";
    int X = 10, Y = 20; // координаты вывода

    public void init() {
    addKeyListener(this);
}

public void keyPressed(KeyEvent ke) {
    showStatus("Key Down");
 // showStatus("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>";
         // msg += "<Стрелка влево>";
            break;
        case KeyEvent.VK_RIGHT:
            msg += "<Right Arrow>";
         // msg += "<Стрелка вправо>";
            break;
    }
    repaint();
}

public void keyReleased(KeyEvent ke) {
    showStatus("Key Up");
    showStatus("Клавиша отпущена");
}
```

```
public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
}

// Отобразить нажатые клавиши.
public void paint(Graphics g) {
    g.drawString(msg, X, Y);
}
```

Пример работы этого аплета показан на рис. 23.3.

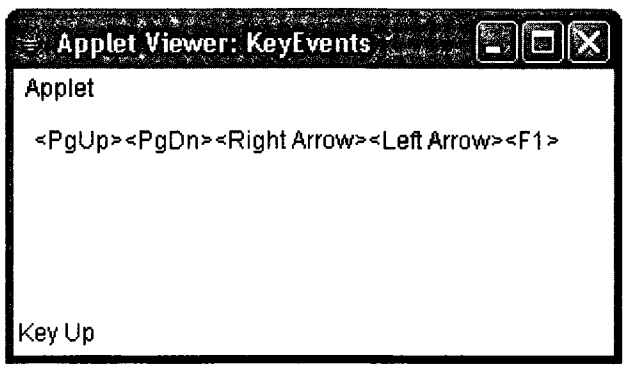


Рис. 23.3. Пример работы аплета, использующего некоторые коды виртуальных клавиш

Процедуры, показанные в приведенных примерах обработки событий мыши и клавиатуры, могут быть обобщены для обработки событий любого типа, включая события элементов управления. В последующих главах вы увидите множество примеров обработки событий других типов, но все они следуют одной и той же базовой структуре, что и только что описанные программы.

Классы адаптеров

В Java имеется специальное средство, называемое *классом адаптера*, который в некоторых ситуациях упрощает реализацию обработчиков событий. Класс адаптера предлагает пустую реализацию всех методов интерфейса слушателя событий. Классы адаптеров удобны, когда вы хотите принимать и обрабатывать только некоторые события, обрабатываемые определенным интерфейсом слушателя. Вы можете определить новый класс для использования в качестве слушателя событий, расширив один из классов адаптеров и реализовав только те события, в которых вы заинтересованы.

Например, класс `MouseMotionAdapter` имеет два метода `mouseDragged()` и `mouseMoved()`, которые определены в интерфейсе `MouseMotionListener`. Если вы заинтересованы только в событиях перетаскивания мыши, можете просто расширить класс `MouseMotionAdapter` и переопределить метод `mouseDragged()`. Пустая реализация метода `mouseMoved()` обработает события перемещения мыши за вас.

В табл. 23.10 перечислены часто используемые классы адаптеров пакета `java.awt.event` и отмечены интерфейсы, реализуемые каждым из них.

Таблица 23.10. Часто используемые интерфейсы слушателей, реализуемые классами адаптеров

Класс адаптера	Интерфейс слушателя
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

В следующем примере демонстрируется применение адаптера. Он отображает сообщение в строке состояния средства просмотра аплетов, когда выполняется щелчок кнопкой мыши или перетаскивание. Однако все прочие события мыши игнорируются. Программа состоит из трех классов. Класс `AdapterDemo` расширяет класс `Applet`. Его метод `init()` создает экземпляр класса `MyMouseAdapter` и регистрирует этот объект для получения уведомлений о событиях мыши. Также он создает экземпляр класса `MyMouseMotionAdapter` и регистрирует его для получения уведомлений о событиях перемещения мыши. Оба конструктора принимают ссылку на аплет в качестве аргумента.

Класс `MyMouseAdapter` расширяет класс `MouseAdapter` и переопределяет метод `mouseClicked()`. Все прочие события мыши игнорируются кодом, унаследованным от класса `MouseAdapter`. Класс `MyMouseMotionAdapter` расширяет класс `MouseMotionAdapter` и переопределяет метод `mouseDragged()`. Другое событие перемещения мыши игнорируется кодом, унаследованным от класса `MouseMotionAdapter`.

Обратите внимание на то, что оба класса слушателей событий сохраняют ссылку на аплет. Эта информация предоставляется в виде аргумента и используется позднее для вызова метода `showStatus()`.

```
// Демонстрация применения адаптера.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/

public class AdapterDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter(this));
        addMouseMotionListener(new MyMouseMotionAdapter(this));
    }
}

class MyMouseAdapter extends MouseAdapter {

    AdapterDemo adapterDemo;
    public MyMouseAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }
    // Обработка щелчка мыши.
    public void mouseClicked(MouseEvent me) {
```

```
        adapterDemo.showStatus("Mouse clicked");
      // adapterDemo.showStatus("Щелчок кнопкой мыши");
    }
}

class MyMouseMotionAdapter extends MouseMotionAdapter {
    AdapterDemo adapterDemo;
    public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }

    // Обработка перетаскивания мыши.
    public void mouseDragged(MouseEvent me) {
        adapterDemo.showStatus("Mouse dragged");
        // adapterDemo.showStatus("Перетаскивание мышью");
    }
}
```

Как видите, отсутствие необходимости реализовать все методы, определенные интерфейсами `MouseMotionListener` и `MouseListener`, позволяет сэкономить значительное количество усилий и избавляет ваш код от перегрузки пустыми методами. В качестве упражнения можете попробовать переписать один из приведенных ранее примеров, обрабатывающих клавиатурный ввод, с использованием класса `KeyAdapter`.

Вложенные классы

Основы вложенных классов были описаны в главе 7. Сейчас вы убедитесь, насколько они важны. Напомним, что *вложенный класс* — это класс, определенный внутри другого класса или даже внутри выражения. В настоящем разделе проиллюстрировано использование вложенных классов для упрощения кода в случае классов адаптеров событий.

Чтобы понять выгоду, которую обеспечивают вложенные классы, рассмотрим аплет, приведенный в следующем листинге. В нем *не используются* вложенные классы. Его назначение — отобразить строку `"Mouse Pressed"` в строке состояния средства просмотра аплетов или браузера, когда нажата кнопка мыши. В этой программе присутствует еще два класса верхнего уровня. Класс `MousePressedDemo` расширяет класс `Applet`, а класс `MyMouseAdapter` — класс `MouseAdapter`. Метод `init()` в классе `MousePressedDemo` создает экземпляр класса `MyMouseAdapter` и предоставляет этот объект в качестве аргумента методу `addMouseListener()`.

Обратите внимание на то, что ссылка на аплет выступает в качестве аргумента конструктора класса `MyMouseAdapter`. Эта ссылка сохраняется в переменной экземпляра для последующего использования методом `mousePressed()`. Когда нажимается кнопка мыши, вызывается метод `showStatus()` аплета через сохраненную ссылку на аплет. Другими словами, метод `showStatus()` вызывается относительно ссылки на аплет, сохраненной в объекте класса `MyMouseAdapter`.

```
// В этом аплете НЕ используются вложенные классы.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="MousePressedDemo" width=200 height=100>
</applet>
*/

public class MousePressedDemo extends Applet {
```

```
    public void init() {
        addMouseListener(new MyMouseAdapter(this));
    }
}

class MyMouseAdapter extends MouseAdapter {
    MousePressedDemo mousePressedDemo;
    public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
        this.mousePressedDemo = mousePressedDemo;
    }
    public void mousePressed(MouseEvent me) {
        mousePressedDemo.showStatus("Mouse Pressed");
     // mousePressedDemo.showStatus("Кнопка мыши нажата");
    }
}
```

В следующем листинге показано, как можно усовершенствовать предыдущую программу, используя вложенный класс. Здесь класс `InnerClassDemo` — это класс верхнего уровня, расширяющий класс `Applet`. Класс `MyMouseAdapter` — это вложенный класс, расширяющий класс `MouseAdapter`. Поскольку класс `MyMouseAdapter` определен внутри области видимости класса `InnerClassDemo`, он имеет доступ ко всем переменным и методам, находящимся в контексте этого класса. Таким образом, метод `mousePressed()` может вызывать метод `showStatus()` непосредственно. Больше нет необходимости делать это через сохраненную ссылку на аплет. А потому не нужно и передавать конструктору `MyMouseAdapter()` ссылку на вызывающий объект.

```
// Демонстрация применения вложенного класса.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200 height=100>
</applet>
*/

public class InnerClassDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter());
    }
    class MyMouseAdapter extends MouseAdapter {
        public void mousePressed(MouseEvent me) {
            showStatus("Mouse Pressed");
         // showStatus("Кнопка мыши нажата");
        }
    }
}
```

Анонимные вложенные классы

Анонимный вложенный класс — это класс, которому не назначено имя. В этом разделе проиллюстрируем, как анонимный вложенный класс может облегчить написание обработчиков событий. Рассмотрим аплет, показанный в следующем листинге. Как и раньше, его назначение — отобразить строку `"Mouse Pressed"` в строке состояния средства просмотра аплетов или браузера, когда нажата кнопка мыши.

```
// Демонстрация применения анонимного вложенного класса.
import java.applet.*;
import java.awt.event.*;
/*
```

```
<applet code="AnonymousInnerClassDemo" width=200 height=100>
</applet>
*/

public class AnonymousInnerClassDemo extends Applet {
    public void init() {
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent me) {
                showStatus("Mouse Pressed");
             // showStatus("Кнопка мыши нажата");
            }
        });
    }
}
```

В этой программе присутствует только один класс верхнего уровня — `AnonymousInnerClassDemo`. Метод `init()` вызывает метод `addMouseListener()`. Его аргументом служит выражение, определяющее и создающее экземпляр анонимного вложенного класса. Давайте тщательно проанализируем это выражение.

Синтаксис `new MouseAdapter() { . . . }` указывает компилятору, что код между фигурными скобками определяет анонимный вложенный класс. Более того, этот класс расширяет класс `MouseAdapter`. Этот новый класс не имеет имени, но его экземпляр автоматически создается при выполнении данного выражения.

Поскольку анонимный вложенный класс определен внутри контекста класса `AnonymousInnerClassDemo`, он имеет доступ ко всем переменным и методам, находящимся в контексте данного класса. Поэтому он может вызывать метод `showStatus()` непосредственно.

Как видите, именованные и анонимные вложенные классы решают некоторые досадные проблемы простым и эффективным способом. Они также позволяют вам создавать более эффективный код.

ГЛАВА 24

Введение в библиотеку AWT: работа с окнами, графикой и текстом

Библиотека AWT (Abstract Window Toolkit) уже была представлена в главе 22, где она использовалась в нескольких примерах аплетов. В этой главе займемся ее более глубоким исследованием. Библиотека AWT включает в себя многочисленные классы и методы, позволяющие создавать окна и управлять ими. Кроме того, она служит фундаментом библиотеки Swing. Библиотека AWT достаточно велика, и полное ее описание легко бы заняло целую книгу. Поэтому невозможно во всех подробностях описать каждый метод, класс или переменную экземпляра библиотеки AWT. Однако в этой и двух последующих главах мы объясним базовые приемы эффективного применения библиотеки AWT для создания аплетов и независимых приложений с графическим интерфейсом пользователя. После этого вы получите возможность самостоятельно поработать и с другими частями библиотеки AWT. Кроме того, вы сможете легко перейти к библиотеке Swing.

В настоящей главе вы научитесь создавать окна, управлять ими, а также шрифтами, выводом текста и графикой. В главе 25 будут описаны различные элементы управления, такие как полосы прокрутки и экранные кнопки, поддерживаемые библиотекой AWT. Также там пойдет речь о дополнительных аспектах механизма обработки событий Java. Глава 26 будет посвящена подсистемам графических изображений и анимации AWT.

Хотя библиотека AWT чаще всего используется в аплетах, она применяется также при создании отдельных окон в среде графического пользовательского интерфейса (GUI), такой как операционная система Windows. По причине согласованности большинство примеров этой главы представлено в виде аплетов. Чтобы запускать их, вам понадобится средство просмотра аплетов или совместимый с Java веб-браузер. Несколько программ, однако, продемонстрируют создание самостоятельных оконных программ.

Прежде чем начать, стоит упомянуть еще об одном. В настоящее время большинство программ на языке Java построено на основе пользовательского интерфейса библиотеки Swing. Поскольку библиотека Swing предлагает более широкие возможности, нежели библиотека AWT, в отношении создания таких распространенных компонентов GUI, как кнопки, окна, списки и флажки, очень легко прийти к мысли, что библиотека AWT утратила свое значение. Однако такое заключение ошибочно. Как уже упоминалось, библиотека Swing построена на базе библиотеки AWT. А потому многие аспекты библиотеки AWT относятся также и к библиотеке Swing. Более того, многие классы библиотеки AWT используются библиотекой Swing, прямо или опосредованно. И наконец, для некоторых типов небольших программ (особенно маленьких аплетов), которые очень ограниченно используют GUI, имеет смысл применять библиотеку AWT вместо библиотеки Swing. Поэтому даже несмотря на то, что большинство интерфейсов в наши дни базируется на библиотеке Swing, хорошее знание библиотеки AWT все еще вос-

требовано. Просто примите к сведению, что без знания библиотеки AWT вы не можете считать себя профессиональным программистом на языке Java.

На заметку! Если вы еще не читали главу 23, сделайте это прямо сейчас. В ней представлен обзор обработки событий, которые будут использоваться во многих примерах настоящей главы.

Классы библиотеки AWT

В пакете `java.awt`, одном из наиболее объемных пакетов Java, содержатся классы библиотеки AWT. К счастью, благодаря логической организации в виде иерархии "сверху вниз", понять и использовать его гораздо легче, чем может показаться на первый взгляд. В табл. 24.1 перечислены некоторые из множества классов AWT.

Таблица 24.1. Примеры классов библиотеки AWT

Класс	Описание
`AWTEvent`	Инкапсулирует события AWT
`AWTEventMulticaster`	Доставляет события множеству слушателей
`BorderLayout`	Граничный диспетчер раскладки. Использует пять компонентов: `North`, `South`, `East`, `West` и `Center`
`Button`	Создает элемент управления — экранную кнопку
`Canvas`	Пустое, свободное от семантики окно
`CardLayout`	Карточный диспетчер раскладки. Эмулирует индексированные карты. Отображается только одна, находящаяся сверху
`Checkbox`	Создает элемент управления — флажок
`CheckboxGroup`	Создает группу флажков
`CheckboxMenuItem`	Создает переключаемый пункт меню
`Choice`	Создает всплывающий список
`Color`	Управляет цветами в переносимой и не зависящей от платформы манере
`Component`	Абстрактный суперкласс для различных компонентов AWT
`Container`	Подкласс класса `Component`, который может содержать другие компоненты
`Cursor`	Инкапсулирует курсор — битовую карту
`Dialog`	Создает диалоговое окно
`Dimension`	Определяет измерения объекта. Ширина сохраняется в `width`, а высота — в `height`
`Event`	Инкапсулирует события
`EventQueue`	Очередь событий. Класс `FileDialog` создает окно, в котором можно выбрать файл
`FlowLayout`	Потоковый диспетчер раскладки. Располагает компоненты слева направо и сверху вниз
`Font`	Инкапсулирует шрифт печати
`FontMetrics`	Инкапсулирует различную информацию, относящуюся к шрифту. Эта информация помогает отображать текст в окне
`Frame`	Создает стандартное окно, снабженное полосой заголовка, элементами управления размерами и полосой меню

Продолжение табл. 24.1

Класс	Описание
`Graphics`	Инкапсулирует графический контекст. Этот контекст используется разнообразными методами вывода для отображения в окне
`GraphicsDevice`	Описывает графическое устройство, такое как экран или принтер
`GraphicsEnvironment`	Описывает коллекцию доступных объектов класса `Font` и `GraphicsDevice`
`GridBagConstraints`	Описывает различные ограничения, относящиеся к классу `GridBagLayout`
`GridBagLayout`	Диспетчер сетчатой управляемой раскладки. Отображает компоненты в соответствии с ограничениями, указанными в объекте класса `GridBagConstraints`
`GridLayout`	Диспетчер сетчатой раскладки. Отображает компоненты в двухмерной сетке
`Image`	Инкапсулирует графические образы
`Insets`	Инкапсулирует границы контейнера
`Label`	Создает метку, отображающую строку
`List`	Создает список, из которого пользователь может выбирать. Аналогичен стандартному окну списка Windows
`MediaTracker`	Управляет медиаобъектами
`Menu`	Создает выпадающее меню
`MenuBar`	Создает полосу меню
`MenuComponent`	Абстрактный класс, реализованный различными классами меню
`MenuItem`	Создает пункт меню
`MenuShortcut`	Инкапсулирует "горячую клавишу" для быстрого вызова пункта меню
`Panel`	Простейший конкретный подкласс класса `Container`
`Point`	Инкапсулирует пару декартовых координат, хранящихся в `x` и `y`
`Polygon`	Инкапсулирует многоугольник
`PopupMenu`	Создает всплывающее меню
`PrintJob`	Абстрактный класс, представляющий задание печати
`Rectangle`	Инкапсулирует прямоугольник
`Robot`	Поддерживает автоматическую проверку приложений на основе библиотеки AWT
`Scrollbar`	Создает элемент управления — полосу прокрутки
`ScrollPane`	Контейнер, предоставляющий горизонтальную и вертикальную полосы прокрутки для другого компонента
`SystemColor`	Содержит цвета для элементов (виджетов) GUI, таких как окна, полосы прокрутки, текст и прочее
`TextArea`	Создает элемент управления — многострочный текстовый редактор

Класс	Описание
TextComponent	Суперкласс для классов TextArea и TextField
TextField	Создает элемент управления — однострочный текстовый редактор
Toolkit	Абстрактный класс, реализованный библиотекой AWT
Window	Создает окно без рамки, без полосы меню и заголовка

Хотя базовая структура библиотеки AWT не менялась со времен Java 1.0, некоторые из оригинальных методов устарели и заменены новыми. Для обеспечения обратной совместимости язык Java все еще поддерживает исходные методы версии 1.0. Однако поскольку эти методы не предназначены для использования в новом коде, мы не станем их описывать в настоящей книге.

Основы окон

Библиотека AWT определяет окна в соответствии с иерархией классов, которые обеспечивают функциональные и специфические возможности на каждом уровне. Два наиболее часто используемых класса окон происходят либо от класса Panel, который используется в аплетах, либо от класса Frame, который создает стандартное окно приложения. Большую часть своих функциональных возможностей эти окна наследуют от своих родительских классов. Поэтому описание иерархии классов, относящихся к этим двум классам, является основополагающим для их понимания. На рис. 24.1 показана иерархия для классов Panel и Frame. Рассмотрим каждый из этих классов.

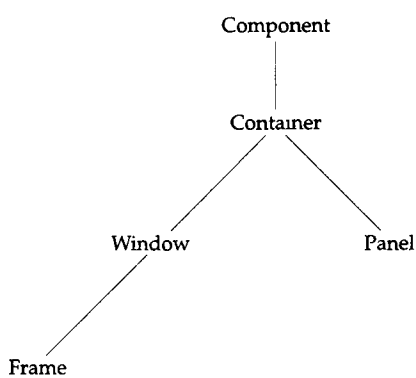


Рис. 24.1. Иерархия классов Panel и Frame

Класс Component

В вершине иерархии находится класс Component. Это — абстрактный класс, инкапсулирующий все атрибуты визуального компонента. За исключением меню, все элементы пользовательского интерфейса являются подклассами класса Component. Он определяет свыше сотни открытых методов, отвечающих за управление событиями, такими как ввод с клавиатуры и события мыши, перемещение и изменение размеров окна, а также перерисовка. (Вы уже использовали многие из

этих методов, когда создавали аплеты в главах 22 и 23.) Объект класса Component отвечает за запоминание текущих цветов фона и переднего плана, а также текущего выбранного шрифта для печати.

Класс Container

Класс Container — это подкласс класса Component. Он имеет дополнительные методы, позволяющие вкладывать в него другие объекты класса Component. Другие объекты класса Container также могут находиться внутри класса Container (поскольку они сами являются экземплярами класса Component), что позволяет строить многоуровневые системы вложенности. Контейнер отвечает за раскладку (т.е. расположение) любых компонентов, которые он содержит.

Это достигается за счет использования различных диспетчеров раскладки, о которых вы узнаете в главе 25.

Класс Panel

Класс Panel — конкретный подкласс класса Container. Таким образом, класс Panel может обеспечивать рекурсивную вложенность и представляет собой конкретный экранный компонент. Класс Panel — суперкласс для класса Applet. Когда экранный вывод направляется в аплет, он рисуется на поверхности объекта класса Panel. По сути, класс Panel — это окно, лишенное полосы заголовка, полосы меню и рамки. И поэтому вы не можете видеть упомянутых элементов, когда аплет запущен внутри браузера. Когда же вы запускаете аплет в средстве просмотра аплетов, то заголовок и рамку предоставляет это средство.

Другие компоненты могут быть добавлены к объекту класса Panel с помощью его метода add() (унаследованного от класса Container). Как только эти компоненты добавлены, вы можете позиционировать их и изменять размеры вручную, используя методы setLocation(), setSize(), setPreferredSize() или setBounds(), определенные в классе Component.

Класс Window

Этот класс создает окно верхнего уровня, которое не содержится внутри другого объекта, а располагается непосредственно на рабочем столе. Обычно вам не придется создавать объекты класса Window непосредственно. Вместо этого вы будете работать с подклассом Frame класса Window.

Класс Frame

Этот класс инкапсулирует то, что обычно воспринимается как "окно". Это подкласс класса Window, имеющий полосу заголовка, полосу меню, границы и элементы управления размером.

Класс Canvas

Не являющийся частью иерархии аплетов или рамочных окон, класс Canvas, возможно, не покажется вам особенно полезным; он инкапсулирует пустое окно, в котором вы можете рисовать. Пример применения класса Canvas будет приведен далее в настоящей книге.

Работа с рамочными окнами

После аплета тип окон, которые вам придется создавать чаще всего, унаследован от класса `Frame`. Вы будете создавать дочерние окна внутри аплетов, а также окна верхнего уровня или дочерние окна для отдельно выполняемых приложений. Как упоминалось, это создает окна в стандартном стиле.

Вот два конструктора класса `Frame`.

```
Frame() throws HeadlessException
Frame(String заголовок) throws HeadlessException
```

Первая форма создает стандартное окно без заголовка. Вторая форма — окно с заголовком, указанным параметром *заголовок*. Обратите внимание на то, что вы не можете задать размеры окна. Вместо этого следует устанавливать размер окна после его создания. Исключение `HeadlessException` передается при попытке создать экземпляр класса `Frame` в среде, которая не поддерживает взаимодействие с пользователем.

Существует несколько ключевых методов, которые вы будете использовать при работе с окнами класса `Frame`. Сейчас мы их и рассмотрим.

Установка размеров окна

Метод `setSize()` используется для установки размеров окна. Его сигнатура показана ниже.

```
void setSize(int новаяШирина, int новаяВысота)
void setSize(Dimension новыйРазмер)
```

Новый размер окна указан параметрами *новаяШирина* и *новаяВысота* или же в полях `width` и `height` объекта класса `Dimension`, переданного в параметре *новыйРазмер*. Размеры задаются в пикселях.

Метод `getSize()` применяется для получения текущего размера окна. Одна из его сигнатур выглядит следующим образом.

```
Dimension getSize()
```

Этот метод возвращает текущий размер окна в полях `width` и `height` объекта класса `Dimension`.

Сокрытие и отображение окна

После того как рамочное окно создано, оно остается невидимым до тех пор, пока вы не вызовите метод `setVisble()`. Его сигнатура такова.

```
void setVisible(boolean флагВидимости)
```

Компонент видим, если этому методу передается аргумент `true`. В противном случае он скрыт.

Установка заголовка окна

Вы можете изменить заголовок рамочного окна, используя метод `setTitle()`, который имеет следующую общую форму.

```
void setTitle(String новыйЗаголовок)
```

Здесь *новыйЗаголовок* — новый заголовок окна.

Закрытие рамочного окна

При использовании рамочного окна ваша программа должна удалять окно с экрана после его закрытия, вызывая метод `setVisible(false)`. Чтобы перехватить событие закрытия окна, следует реализовать метод `windowClosing()` интерфейса `WindowListener`. Внутри метода `windowClosing()` следует удалить окно с экрана. Пример, приведенный в следующем разделе, иллюстрирует этот прием.

Создание рамочного окна в аплете

Хотя получить окно можно, создав экземпляр класса `Frame`, вам редко доведется поступать так, поскольку с таким окном не так много можно делать. Например, вы не сможете принимать или обрабатывать события, которые происходят в нем, или просто выводить в него информацию. В основном, вам придется создавать подклассы класса `Frame`. Это позволит переопределять методы класса `Frame` и обеспечивать обработку событий.

Создать новое рамочное окно аплета достаточно просто. Для начала создается подкласс класса `Frame`. Затем переопределяется любой стандартный метод аплета, такой как `init()`, `start()` и `stop()`, чтобы отображать или скрывать фрейм по необходимости. И наконец, реализуется метод `windowClosing()`.

Как только определите подкласс класса `Frame`, сможете создать объект этого класса. Это приведет к появлению рамочного окна, которое, однако, изначально будет невидимым. Вы делаете его видимым при помощи вызова метода `setVisible()`. При создании окно получает высоту и ширину по умолчанию. Вы можете установить размеры окна явно с помощью метода `setSize()`.

Приведенный ниже аплет создает подкласс класса `Frame` по имени `SampleFrame`. Экземпляр окна этого класса создается внутри метода `init()` класса `AppletFrame`. Обратите внимание на то, что класс `SampleFrame` вызывает конструктор класса `Frame`. Это позволяет создать стандартное рамочное окно с заголовком, переданным в параметре `title`. Этот пример переопределяет методы `start()` и `stop()` аплета так, что они, соответственно, отображают и скрывают дочернее окно. Это позволяет автоматически удалить окно, когда вы прерываете работу аплета, когда закрываете окно или, при использовании браузера, когда переходите на другую страницу. Это также вызывает отображение дочернего окна, когда браузер возвращается к аплету.

```
// Создание дочернего окна из аплета.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*
<applet code="AppletFrame" width=300 height=50>
</applet>
*/

// Создание подкласса Frame.
    class SampleFrame extends Frame {
        SampleFrame(String title) {
        super(title);

        // создание объекта для обработки событий окна
        MyWindowAdapter adapter = new MyWindowAdapter(this);

        // регистрация его для получения событий
```

```
        addWindowListener(adapter);
    }

    public void paint(Graphics g) {
        g.drawString("This is in frame window", 10, 40);
    }
}

class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;

    public MyWindowAdapter(SampleFrame sampleFrame) {
        this.sampleFrame = sampleFrame;
    }

    public void windowClosing(WindowEvent we) {
        sampleFrame.setVisible(false);
    }
}

// Создание рамочного окна.
public class AppletFrame extends Applet {
    Frame f;

    public void init() {
        f = new SampleFrame("A Frame Window");
        f.setSize(250, 250);
        f.setVisible(true);
    }

    public void start() {
        f.setVisible(true);
    }

    public void stop() {
        f.setVisible(false);
    }

    public void paint(Graphics g) {
        g.drawString("Это окно аплета", 10, 20);
    }
}
```

Результат выполнения этого аплета показан на рис. 24.2.

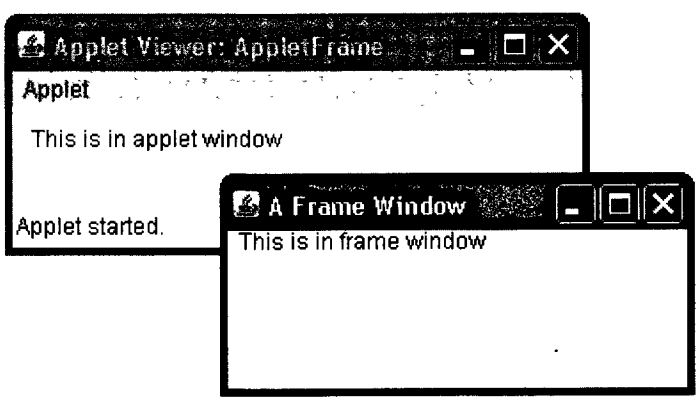


Рис. 24.2. Создание дочернего окна из аплета

Обработка событий в рамочном окне

Поскольку класс Frame — подкласс класса Component, он наследует все, что определено в классе Component. Это значит, что вы можете использовать рамочное окно и управлять им точно так же, как делаете это с главным окном аплета. Например, вы можете переопределить метод paint() для отображения вывода, вызвать метод repaint(), когда вам нужно восстановить окно, и добавить обработчики событий. Всякий раз, когда в окне происходят события, вызываются обработчики, определенные этим окном. Каждое окно обрабатывает свои собственные события. Например, следующая программа создает окно, реагирующее на события мыши. Главное окно аплета также реагирует на события мыши. Если вы поэкспериментируете с этой программой, то увидите, что события мыши посылаются окну, в котором они происходят.

```
// Обработка событий мыши в дочернем окне и окне аплета.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="WindowEvents" width=300 height=50>
</applet>
*/

// Создание подкласса Frame.
class SampleFrame extends Frame
implements MouseListener, MouseMotionListener {

    String msg = "";
    int mouseX=10, mouseY=40;
    int movX=0, movY=0;

    SampleFrame(String title) {
    super(title);
    // зарегистрировать объект для получения его собственных событий мыши
    addMouseListener(this);
    addMouseMotionListener(this);
    // создать объект для обработки событий окна
    MyWindowAdapter adapter = new MyWindowAdapter(this);
    // зарегистрировать для получения этих событий
    addWindowListener(adapter);
}

// Обработка щелчка мыши.
    public void mouseClicked(MouseEvent me) {
    }

    // Обработка входа курсора мыши.
    public void mouseEntered(MouseEvent evtObj) {
        // сохранить координаты
        mouseX = 10;
        mouseY = 54;
        msg = "Mouse just entered child.";
     // msg = "Курсор мыши только что вошел в дочернее окно.";
        repaint();
    }

    // Обработка выхода курсора мыши.
    public void mouseExited(MouseEvent evtObj) {
        // сохранить координаты
        mouseX = 10;
        mouseY = 54;
```

```
        msg = "Mouse just left child window.";
      // msg = "Курсор мыши только что покинул дочернее окно.";
        repaint();
    }

    // Обработка нажатия кнопки мыши.
    public void mousePressed(MouseEvent me) {
        // сохранить координаты
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Down";
        repaint();
    }

    // Обработка отпускания кнопки мыши.
    public void mouseReleased(MouseEvent me) {
        // сохранить координаты
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Up";
        repaint();
    }

    // Обработка перетаскивания мышью.
    public void mouseDragged(MouseEvent me) {
        // сохранить координаты
        mouseX = me.getX();
        mouseY = me.getY();
        movX = me.getX();
        movY = me.getY();
        msg = "*";
        repaint();
    }

    // Обработка перемещения мыши.
    public void mouseMoved(MouseEvent me) {
        // сохранить координаты
        movX = me.getX();
        movY = me.getY();
        repaint(0, 0, 100, 60);
    }

    public void paint(Graphics g) {
        g.drawString(msg, mouseX, mouseY);
        g.drawString("Mouse at " + movX + ", " + movY, 10, 40);
      // g.drawString("Курсор мыши в " + movX + ", " + movY, 10, 40);
    }
}

class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;
    public MyWindowAdapter(SampleFrame sampleFrame) {
        this.sampleFrame = sampleFrame;
    }
    public void windowClosing(WindowEvent we) {
        sampleFrame.setVisible(false);
    }
}

// Окно аплета.
public class WindowEvents extends Applet
implements MouseListener, MouseMotionListener {
    SampleFrame f;
```

```
String msg = "";
int mouseX=0, mouseY=10;
int movX=0, movY=0;

// Создание рамочного окна.
public void init() {
    f = new SampleFrame("Handle Mouse Events");
    f.setSize(300, 200);
    f.setVisible(true);
    // зарегистрировать объект для получения его собственных
    // событий мыши
    addMouseListener(this);
    addMouseMotionListener(this);
}

// Удаление рамочного окна при останове аплета.
public void stop() {
    f.setVisible(false);
}

// Показать рамочное окно при запуске аплета.
public void start() {
    f.setVisible(true);
}

// Обработка щелчка мыши.
public void mouseClicked(MouseEvent me) {
}

// Обработка входа мыши.
public void mouseEntered(MouseEvent me) {
    // сохранить координаты
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just entered applet window.";
 // msg = "Курсор мыши только что вошел в окно аплета.";
    repaint();
}

// Обработка выхода мыши.
public void mouseExited(MouseEvent me) {
    // сохранить координаты
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just left applet window.";
 // msg = "Курсор мыши только что покинул окно аплета.";
    repaint();
}

// Обработка нажатия кнопки мыши.
public void mousePressed(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
}

// Обработка отпускания кнопки мыши.
public void mouseReleased(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
```

```
        msg = "Up";
        repaint();
    }

    // Обработка перетаскивания мышью.
    public void mouseDragged(MouseEvent me) {
        // сохранить координаты
        mouseX = me.getX();
        mouseY = me.getY();
        movX = me.getX();
        movY = me.getY();
        msg = "*";
        repaint();
    }

    // Обработка перемещения мыши.
    public void mouseMoved(MouseEvent me) {
        // сохранить координаты
        movX = me.getX();
        movY = me.getY();
        repaint(0, 0, 100, 20);
    }

    // Отображение сообщения в окне аплета
    public void paint(Graphics g) {
        g.drawString(msg, mouseX, mouseY);
        g.drawString("Mouse at " + movX + ", " + movY, 0, 10);
     // g.drawString("Курсор мыши в " + movX + ", " + movY, 0, 10);
    }
}
```

Результат выполнения этого аплета показан на рис. 24.3.

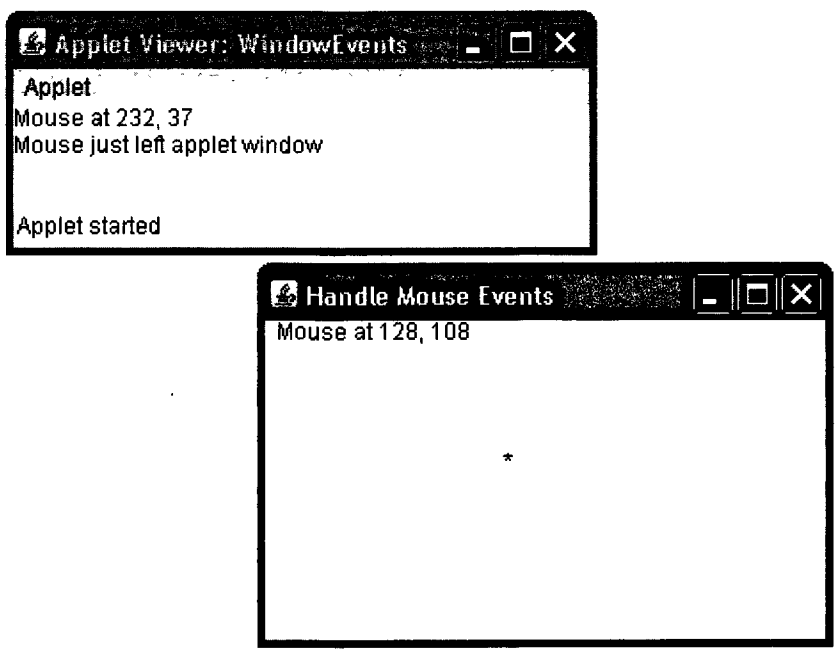


Рис. 24.3. Обработка событий мыши в дочернем окне и окне аплета

Создание оконной программы

Хотя создание аплетов — привычная задача для применения библиотеки AWT языка Java, на основе библиотеки AWT также можно создавать автономные приложения. Для этого просто создайте экземпляр нужного окна или окон внутри метода `main()`. Например, следующая программа создает рамочное окно, реагирующее на щелчки мыши и нажатия клавиш.

```
// Создание приложения на базе AWT.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

// Создание рамочного окна.
public class AppWindow extends Frame {
    String keymsg = "This is a test.";
    String mousemsg = "";
    int mouseX=30, mouseY=30;

    public AppWindow() {
        addKeyListener(new MyKeyAdapter(this));
        addMouseListener(new MyMouseAdapter(this));
        addWindowListener(new MyWindowAdapter());
    }

    public void paint(Graphics g) {
        g.drawString(keymsg, 10, 40);
        g.drawString(mousemsg, mouseX, mouseY);
    }

    // Создание окна.
    public static void main(String args[]) {
        AppWindow appwin = new AppWindow();

        appwin.setSize(new Dimension(300, 200));
        appwin.setTitle("An AWT-Based Application");
     // appwin.setTitle("Приложение AWT");
        appwin.setVisible(true);
    }
}

class MyKeyAdapter extends KeyAdapter {
    AppWindow appWindow;
    public MyKeyAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }

    public void keyTyped(KeyEvent ke) {
        appWindow.keymsg += ke.getKeyChar();
        appWindow.repaint();
    };
}

class MyMouseAdapter extends MouseAdapter {
    AppWindow appWindow;
    public MyMouseAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }

    public void mousePressed(MouseEvent me) {
        appWindow.mouseX = me.getX();
        appWindow.mouseY = me.getY();
```

```
            appWindow.mousemsg = "Mouse Down at " + appWindow.mouseX +
        // appWindow.mousemsg = "Клавиша мыши нажата в " +
        //                       appWindow.mouseX +
                                 ", " + appWindow.mouseY;
            appWindow.repaint();
        }
}

class MyWindowAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent we) {
        System.exit(0);
    }
}
```

Результат выполнения этого аплета показан на рис. 24.4.

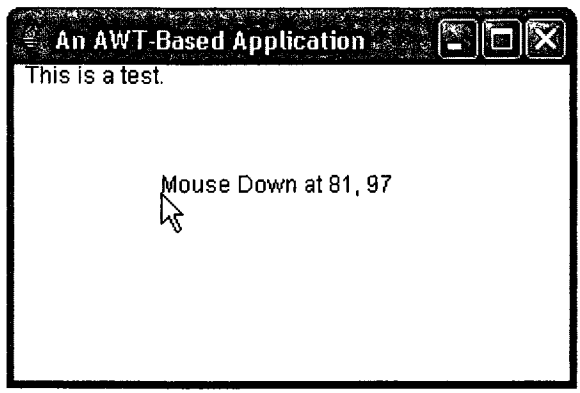


Рис. 24.4. Пример приложения AWT

Однажды созданное рамочное окно начинает "жить собственной жизнью". Обратите внимание на то, что метод `main()` завершается вызовом метода `appwin.setVisible(true)`. Однако программа продолжает выполняться до тех пор, пока вы не закроете окно. По сути, при создании оконного приложения вы используете метод `main()` для запуска окна верхнего уровня. После этого ваша программа функционирует как приложение на основе GUI, а не как консольное приложение вроде показанных ранее.

Отображение информации внутри окна

В наиболее общем случае окно является контейнером для информации. Хотя в предыдущих примерах мы уже выводили небольшие объемы текста в окно, все же до сих пор мы не использовали возможности окон представлять высококачественный текст и графику. На самом деле большая часть мощи библиотеки AWT проявляется именно благодаря такой поддержке. Поэтому далее в этой главе мы обсудим возможности языка Java по выводу текста, графики и управлению шрифтами. Как вы убедитесь, они обеспечивают как необходимую мощность, так и гибкость.

Работа с графикой

Библиотека AWT предоставляет богатый ассортимент графических методов. Весь графический вывод осуществляется относительно окна. Это может быть

главное окно аплета, дочернее окно аплета или же окно автономного приложения. Начальная точка каждого окна находится в верхнем левом углу и имеет координаты 0,0. Координаты указываются в пикселях. Весь вывод в окно выполняется в конкретном графическом контексте. *Графический контекст*, инкапсулированный классом `Graphics`, получаем двумя способами:

- передается в метод, такой как `paint()` или `update()`, как аргумент;
- возвращается методом `getGraphics()` класса `Component`.

Для обеспечения согласованности далее продемонстрирует графику в главном окне аплета. Однако те же приемы применимы к любому окну.

Класс `Graphics` определяет множество функций рисования. Любая фигура может быть нарисована только контуром либо залита. Объекты рисуются и заливаются текущим выбранным цветом графики, которым по умолчанию является черный. При рисовании графического объекта, выходящего за пределы окна, его вывод автоматически усекается. Давайте рассмотрим несколько методов рисования.

Рисование линий

Линии рисуются с помощью метода `drawLine()`, показанного ниже.

```
void drawLine(int началоX, int началоY, int конецX, int конецY)
```

Метод `drawLine()` отображает линию в текущем цвете рисования, начиная с точки *началоX*, *началоY* и заканчивая *конецX*, *конецY*.

Следующий аплет рисует несколько линий.

```
// Рисование линий
import java.awt.*;
import java.applet.*;
/*
<applet code="Lines" width=300 height=200>
</applet>
*/

public class Lines extends Applet {
    public void paint(Graphics g) {
        g.drawLine(0, 0, 100, 100);
        g.drawLine(0, 100, 100, 0);
        g.drawLine(40, 25, 250, 180);
        g.drawLine(75, 90, 400, 400);
        g.drawLine(20, 150, 400, 40);
        g.drawLine(5, 290, 80, 19);
    }
}
```

Результат выполнения этого аплета показан на рис. 24.5.

Рисование прямоугольников

Методы `drawRect()` и `fillRect()` отображают, соответственно, контурный и закрашенный прямоугольники.

Выглядят они так.

```
void drawRect(int сверху, int слева, int ширина, int высота)
void fillRect(int сверху, int слева, int ширина, int высота)
```

Левый верхний угол прямоугольника расположен в *сверху*, *слева*. Размеры прямоугольника задаются параметрами *ширина* и *высота*.

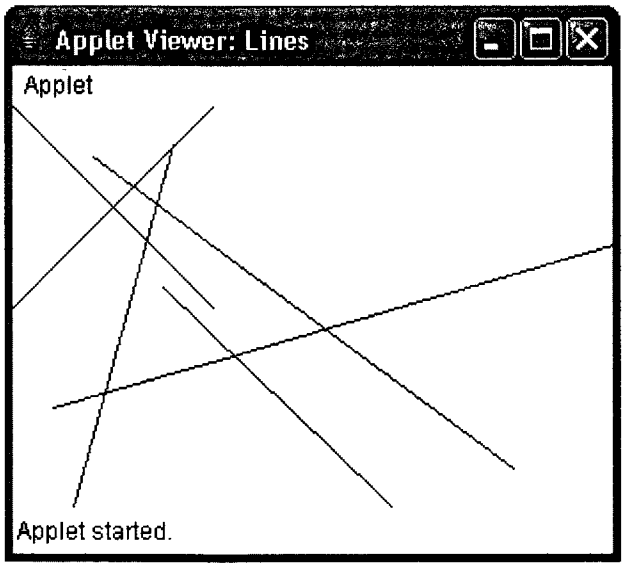


Рис. 24.5. Пример рисования линий

Чтобы нарисовать прямоугольник со скругленными углами, используйте методы `drawRoundRect()` и `fillRoundRect()`.

```
void drawRoundRect(int сверху, int слева, int ширина, int высота,
                   int диаметрX, int диаметрY)
void fillRoundRect(int сверху, int слева, int ширина, int высота,
                   int диаметрX, int диаметрY)
```

Нарисованные этими методами прямоугольники будут иметь скругленные углы. Диаметр скругления по оси X указывается параметром *диаметрX*. Диаметр скругления дуги по оси Y задается в *диаметрY*.

Приведенный ниже аплет рисует несколько прямоугольников.

```
// Рисование прямоугольников
import java.awt.*;
import java.applet.*;
/*
<applet code="Rectangles" width=300 height=200>
</applet>
*/

public class Rectangles extends Applet {
    public void paint(Graphics g) {
        g.drawRect(10, 10, 60, 50);
        g.fillRect(100, 10, 60, 50);
        g.drawRoundRect(190, 10, 60, 50, 15, 15);
        g.fillRoundRect(70, 90, 140, 100, 30, 40);
    }
}
```

Результат выполнения этого аплета показан на рис. 24.6.

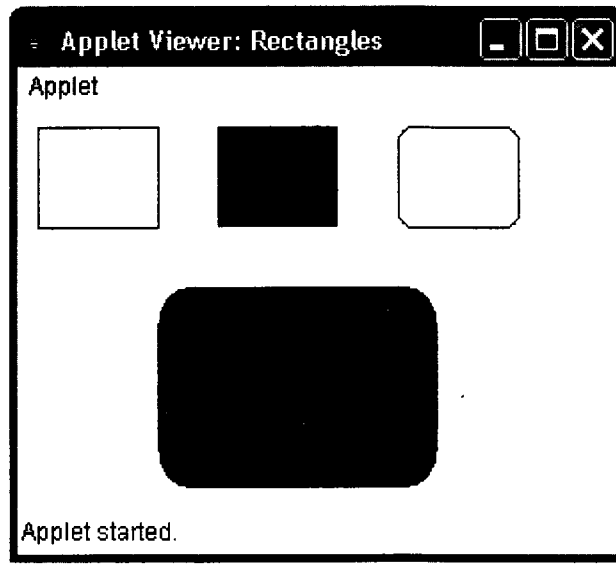


Рис. 24.6. Пример рисования прямоугольников

Рисование эллипсов и окружностей

Чтобы нарисовать эллипс, используйте метод `drawOval()`. Для его заливки применяйте метод `fillOval()`. Эти методы выглядят следующим образом.

```
void drawOval(int сверху, int слева, int ширина, int высота)
void fillOval(int сверху, int слева, int ширина, int высота)
```

Эллипс рисуется внутри описанного прямоугольника, верхний левый угол которого имеет координаты *сверху*, *слева*, а ширина и высота указаны в *ширина* и *высота*. Чтобы нарисовать круг, в качестве описанного прямоугольника задавайте квадрат.

Следующая программа рисует несколько эллипсов.

```
// Рисование эллипсов.
import java.awt.*;
import java.applet.*;
/*
<applet code="Ellipses" width=300 height=200>
</applet>
*/

public class Ellipses extends Applet {
    public void paint(Graphics g) {
        g.drawOval(10, 10, 50, 50);
        g.fillOval(100, 10, 75, 50);
        g.drawOval(190, 10, 90, 30);
        g.fillOval(70, 90, 140, 100);
    }
}
```

Результат выполнения этого аплета показан на рис. 24.7.

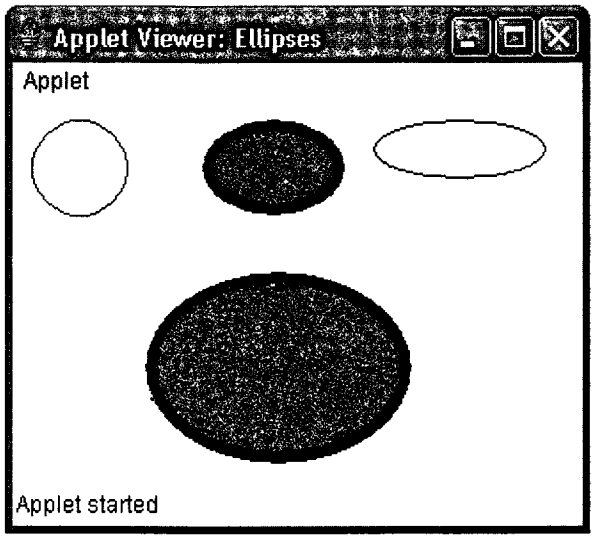


Рис. 24.7. Пример рисования эллипсов

Рисование дуг

Дуги могут быть нарисованы методами `drawArc()` и `fillArc()`.

```
void drawArc(int сверху, int слева, int ширина, int высота,
             int начУгол, int уголПоворота)
void fillArc(int сверху, int слева, int ширина, int высота,
             int начУгол, int уголПоворота)
```

Дуга ограничена прямоугольником, верхний левый угол которого находится в точке с координатами *сверху*, *слева*, а ширина и высота указаны параметрами *ширина* и *высота*. Дуга рисуется начиная с угла *начУгол* и продолжается на величину угла *уголПоворота*. Углы указываются в градусах. Нуль градусов соответствует горизонтали, направленной по часовой стрелке, показывающей на три часа. Дуга рисуется в направлении против часовой стрелки, если значение *уголПоворота* положительно, и по часовой стрелке – если значение *уголПоворота* отрицательно. Таким образом, чтобы нарисовать дугу от 12 часов до 6 часов, нужно указать в качестве начального угла 90 градусов, а угла поворота – 180 градусов.

```
// Рисование дуг окружностей.
import java.awt.*;
import java.applet.*;
/*
<applet code="Arcs" width=300 height=200>
</applet>
*/

public class Arcs extends Applet {
    public void paint(Graphics g) {
        g.drawArc(10, 40, 70, 70, 0, 75);
        g.fillArc(100, 40, 70, 70, 0, 75);
        g.drawArc(10, 100, 70, 80, 0, 175);
        g.fillArc(100, 100, 70, 90, 0, 270);
        g.drawArc(200, 80, 80, 80, 0, 180);
    }
}
```

Результат выполнения этого аплета показан на рис. 24.8.

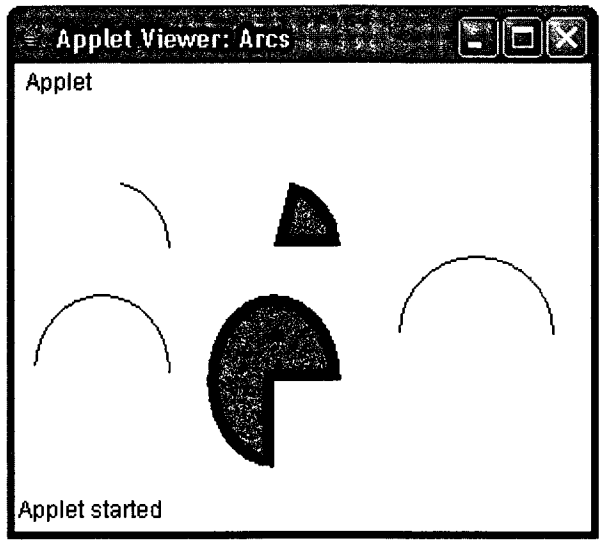


Рис. 24.8. Пример рисования дуг окружностей

Рисование многоугольников

Можно рисовать фигуры произвольной формы, используя методы `drawPolygon()` и `fillPolygon()`, показанные ниже.

```
void drawPolygon(int x[ ], int y[ ], int колТочек)
void fillPolygon(int x[ ], int y[ ], int колТочек)
```

Конечные точки многоугольника указываются парами координат в массивах *x* и *y*. Количество точек, определенных параметрами *x* и *y*, указывается параметром *колТочек*. Существуют альтернативные формы этих методов, в которых многоугольник определяется объектом класса `Polygon`.

Следующий аплет рисует форму песочных часов.

```
// Рисование многоугольника.
import java.awt.*;
import java.applet.*;
/*
<applet code="HourGlass" width=230 height=210>
</applet>
*/

public class HourGlass extends Applet {
    public void paint(Graphics g) {
        int xpoints[] = {30, 200, 30, 200, 30};
        int ypoints[] = {30, 30, 200, 200, 30};
        int num = 5;
        g.drawPolygon(xpoints, ypoints, num);
    }
}
```

Результат выполнения этого аплета показан на рис. 24.9.

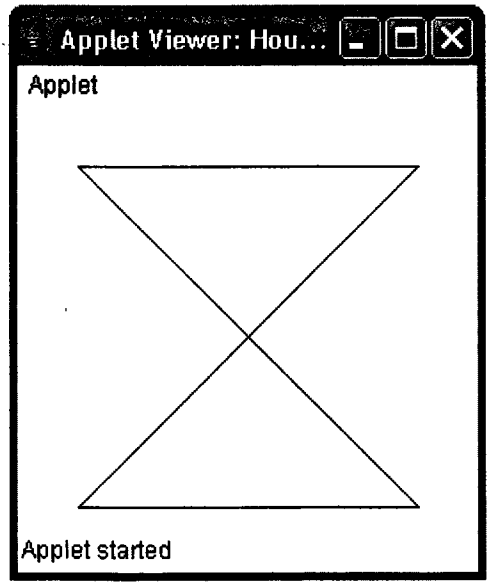


Рис. 24.9. Пример рисования многоугольника

Установка размеров графики

Нередко возникает необходимость установить такой размер графического объекта, чтобы он занял текущий размер окна, в котором он нарисован. Чтобы добиться этого, получите текущие размеры окна вызовом метода `getSize()` для объекта окна. Этот метод вернет размеры окна, инкапсулируя их в объекте класса `Dimension`. Получив текущий размер окна, вы можете соответствующим образом масштабировать графику.

Для демонстрации этого приема рассмотрим аплет, который, начиная с квадрата размером 200×200 пикселей, будет увеличивать его с каждым щелчком мыши на 25 пикселей до тех пор, пока тот не получит размер 500×500 пикселей. В этой точке следующий щелчок вернет его к размеру 200×200 пикселей, после чего процесс начнется сначала.

Внутри окна прямоугольник рисуется вдоль внутренних границ окна; внутри прямоугольника рисуется символ X, чтобы заполнить его. Этот аплет работает в приложении `appletviewer`, но может не работать в окне браузера.

```
// Изменение размеров для заполнения текущего размера окна.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code="ResizeMe" width=200 height=200>
</applet>
*/

public class ResizeMe extends Applet {
    final int inc = 25;
    int max = 500;
    int min = 200;
    Dimension d;
```

```
    public ResizeMe() {
        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));
            }
        });
    }

    public void paint(Graphics g) {
        d = getSize();
        g.drawLine(0, 0, d.width-1, d.height-1);
        g.drawLine(0, d.height-1, d.width-1, 0);
        g.drawRect(0, 0, d.width-1, d.height-1);
    }
}
```

Работа с цветом

Язык Java поддерживает цвета в переносимой, независимой от устройства манере. Система цветов библиотеки AWT позволяет задать любой цвет по вашему желанию. Затем она находит наилучшее соответствие этому цвету, учитывая ограничения оборудования дисплея, на котором выполняется ваш аплет или программа. Таким образом, код не должен заботиться о различиях в поддержке цвета различными аппаратными устройствами. Цвет инкапсулирован в классе `Color`.

Как вы уже видели в главе 22, класс `Color` определяет несколько констант (вроде `Color.black`) для описания множества наиболее часто используемых цветов. Вы можете также создавать свои собственные цвета, используя один из доступных конструкторов цвета. Ниже приведены три наиболее часто используемые формы таких конструкторов.

```
Color(int red, int green, int blue)
Color(int значениеRGB)
Color(float red, float green, float blue)
```

Первый конструктор принимает три аргумента, задающие цвет как смесь красной, зеленой и синей составляющих. Эти значения должны находиться в пределах от 0 до 255, как в следующем примере.

```
new Color(255, 100, 100); // светло-красный
```

Второй конструктор цвета принимает единственный целочисленный аргумент, содержащий смесь интенсивностей красного, зеленого и синего, упакованную в одно целое число. Это целое число организовано так, что красная составляющая упакована в биты с 16 по 23, зеленая — с 8 по 15 и синяя — с 0 по 7. Вот пример применения этого конструктора.

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color darkRed = new Color(newRed);
```

И последний конструктор, `Color(float, float, float)`, принимает три значения с плавающей точкой (в диапазоне от 0,0 до 1,0), указывающие относительные значения смеси красного, зеленого и синего.

Создав подобным образом цвет, вы можете использовать его для установки цвета переднего плана и/или фона с помощью методов `setForeground()` и `setBackground()`, описанных в главе 22. Также вы можете выбрать его в качестве текущего цвета рисования.

Методы класса Color

Класс Color определяет несколько методов, помогающих манипулировать цветами. Рассмотрим некоторые из них.

Использование цвета, насыщенности и яркости

Цветовая модель "*цвет-насыщенность-яркость*" (hue-saturation-brightness – HSB) представляет собой альтернативу модели "красный-зеленый-синий" (red-green-blue – RGB) для указания определенного цвета. Образно говоря, *цвет* в модели HSB представляет собой цветовой круг. Он может быть задан числом в диапазоне от 0,0 до 1,0, приближенно образуя радугу. (Ее основными цветами будут красный, оранжевый, желтый зеленый, синий, индиго и фиолетовый.) *Насыщенность* – вторая шкала в диапазоне от 0,0 до 1,0 – представляет примеси для интенсивности цвета. Значения *яркости* также лежат в диапазоне от 0,0 до 1,0, где 1 – ярко-белый, а 0 – черный. Класс Color поддерживает два метода, позволяющие выполнять преобразования между моделями RGB и HSB; методы показаны ниже.

```
static int HSBtoRGB(float hue, float saturation, float brightness)
static float[ ] RGBtoHSB(int red, int green, int blue, float значения[ ])
```

Метод HSBtoRGB() возвращает упакованное значение RGB, совместимое с конструктором Color(int). Метод RGBtoHSB() возвращает массив чисел с плавающей точкой, представляющих значения HSB, соответствующие составляющим RGB. Если массив *значения* не пуст, то он представляет заданные значения HSB и возвращается. В противном случае создается новый массив, и значения HSB возвращаются в нем. В любом случае массив содержит цвет в элементе с индексом 0, насыщенность – в элементе 1 и яркость – в элементе 2.

Методы getRed(), getGreen(), getBlue()

Вы можете получить красную, зеленую и синюю составляющие цвета независимо, используя для этого методы getRed(), getGreen() и getBlue(), показанные ниже.

```
int getRed()
int getGreen()
int getBlue()
```

Каждый из этих методов возвращает компонент цвета RGB, извлеченный из вызывающего объекта класса Color в нижних 8 бит целого числа.

Метод getRgb()

Чтобы получить упакованное представление цвета RGB, предусмотрен метод getRed(), показанный ниже.

```
int getRGB()
```

Возвращаемое значение организовано, как было описано выше.

Установка текущего цвета графики

По умолчанию графические объекты рисуются текущим цветом переднего плана. Вы можете изменить этот цвет вызовом метода setColor() класса Graphics.

```
void setColor(Color новыйЦвет)
```

Здесь *новыйЦвет* определяет новый цвет рисования. Вы можете получить текущий установленный цвет, вызвав метод getColor(), показанный ниже.

```
Color getColor()
```

Аплет, демонстрирующий цвета

Следующий аплет создает несколько цветов и рисует различные объекты, используя эти цвета.

```
// Демонстрация цветов.
import java.awt.*;
import java.applet.*;
/*
<applet code="ColorDemo" width=300 height=200>
</applet>
*/

public class ColorDemo extends Applet {
    // рисование линий
    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(0, 0, 100, 100);
        g.drawLine(0, 100, 100, 0);

        g.setColor(c2);
        g.drawLine(40, 25, 250, 180);
        g.drawLine(75, 90, 400, 400);

        g.setColor(c3);
        g.drawLine(20, 150, 400, 40);
        g.drawLine(5, 290, 80, 19);

        g.setColor(Color.red);
        g.drawOval(10, 10, 50, 50);
        g.fillOval(70, 90, 140, 100);

        g.setColor(Color.blue);
        g.drawOval(190, 10, 90, 30);
        g.drawRect(10, 10, 60, 50);

        g.setColor(Color.cyan);
        g.fillRect(100, 10, 60, 50);
        g.drawRoundRect(190, 10, 60, 50, 15, 15);
    }
}
```

Установка режима рисования

Режим рисования (paint mode) определяет, как рисуются объекты в окне. По умолчанию новый вывод в окно перекрывает любое существующее содержимое. Однако можно получить новые объекты, объединенные оператором XOR с предыдущим содержимым окна, используя метод `setXORMode()`, как показано ниже.

```
void setXORMode(Color цветXOR)
```

Здесь *цветXOR* определяет цвет, который будет соединен оператором XOR с содержимым окна во время рисования. Преимущество режима XOR в том, что новый объект всегда будет гарантированно видимым, независимо от того, какого цвета был объект, нарисованный ранее.

Чтобы вернуться к методу рисования с перекрытием, вызовите метод setPaintMode().

```
void setPaintMode()
```

Обычно метод перекрытия используется для нормального вывода, а режим XOR – для специальных целей. Например, следующая программа отображает крестообразный "прицел", отслеживающий курсор мыши. Крест отображается рисованием в режиме XOR в поле окна и потому всегда видим, независимо от цвета фона.

```
// Демонстрация применения режима XOR.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="XOR" width=400 height=200>
</applet>
*/

public class XOR extends Applet {
    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();
            }
        });
    }

    public void paint(Graphics g) {
        g.drawLine(0, 0, 100, 100);
        g.drawLine(0, 100, 100, 0);
        g.setColor(Color.blue);
        g.drawLine(40, 25, 250, 180);
        g.drawLine(75, 90, 400, 400);
        g.setColor(Color.green);
        g.drawRect(10, 10, 60, 50);
        g.fillRect(100, 10, 60, 50);
        g.setColor(Color.red);
        g.drawRoundRect(190, 10, 60, 50, 15, 15);
        g.fillRoundRect(70, 90, 140, 100, 30, 40);
        g.setColor(Color.cyan);
        g.drawLine(20, 150, 400, 40);
        g.drawLine(5, 290, 80, 19);

        // xor крестообразный "прицел"
        g.setXORMode(Color.black);
        g.drawLine(chsX-10, chsY, chsX+10, chsY);
        g.drawLine(chsX, chsY-10, chsX, chsY+10);
        g.setPaintMode();
    }
}
```

Пример вывода этого аплета показан на рис. 24.10.

Работа со шрифтами

Библиотека AWT поддерживает множество типов шрифтов. Ранее шрифты из области традиционной печати были важной частью создаваемых компьютерами документов и представлений. Гибкость библиотеки AWT обеспечивается за счет абстрагирования операций манипулирования шрифтами и возможности динамического выбора шрифтов.

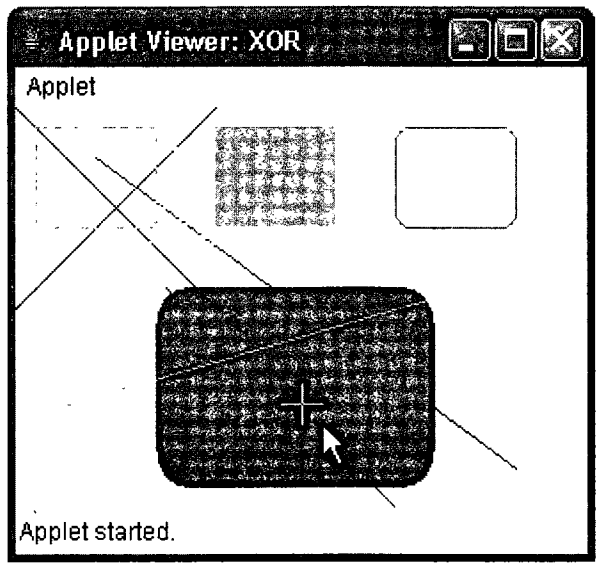


Рис. 24.10. Пример применения режима XOR

Шрифты имеют имя семейства, логическое имя шрифта и название гарнитуры. *Имя семейства* – обобщенное имя шрифта, такое как Courier. *Логическое имя* определяет категорию шрифтов вроде Monospaced. *Гарнитура* определяет конкретный шрифт, такой как Courier Italic.

Шрифты инкапсулируются классом `Font`. Несколько методов, определенных в классе `Font`, перечислено в табл. 24.2.

Таблица 24.2. Методы, определенные в классе `Font`

Метод	Описание
`static Font decode(String` *строка*`)`	Возвращает шрифт по заданному имени
`boolean equals(Object` *объектШрифта*`)`	Возвращает значение `true`, если вызывающий объект содержит тот же шрифт, что и указанный в *объектШрифта*
`String getFamily()`	Возвращает имя семейства шрифтов, к которому относится вызывающий шрифт
`static Font getFont(String` *свойство*`)`	Возвращает шрифт, ассоциированный с системным свойством, указанным в *свойство*. Если свойство *свойство* не существует, возвращается значение `null`
`static Font getFont(String` *свойство*`, Font` *стандартныйШрифт*`)`	Возвращает шрифт, ассоциированный с системным свойством, указанным параметром *свойство*. Если *свойство* не существует, возвращается *стандартныйШрифт*

Окончание табл. 24.2

Метод	Описание
`String getFontName()`	Возвращает название гарнитуры вызывающего шрифта
`String getName()`	Возвращает логическое имя вызывающего шрифта
`int getSize()`	Возвращает размер в точках вызывающего шрифта
`int getStyle()`	Возвращает значения стиля вызывающего шрифта
`int hashCode()`	Возвращает хеш-код, ассоциированный с вызывающим объектом
`boolean isBold()`	Возвращает значение `true`, если шрифт включает значение стиля `BOLD`. В противном случае возвращается значение `false`
`boolean isItalic()`	Возвращает значение `true`, если шрифт включает значение стиля `ITALIC`. В противном случае возвращается значение `false`
`boolean isPlain()`	Возвращает значение `true`, если шрифт включает значение стиля `PLAIN`. В противном случае возвращается значение `false`
`String toString()`	Возвращает строковый эквивалент вызывающего шрифта

В классе `Font` определены переменные, которые описаны в табл. 24.3.

Таблица 24.3. Переменные, определенные в классе `Font`

Переменная	Значение
`String name`	Имя шрифта
`float pointSize`	Размер шрифта в точках
`int size`	Размер шрифта в точках
`int style`	Стиль шрифта

Определение доступных шрифтов

При работе со шрифтами зачастую нужно знать, какие шрифты присутствуют на конкретной машине. Для получения этой информации служит метод `getAvailableFontFamilyNames()`, определенный в классе `GraphicsEnvironment`.

```
String[] getAvailableFontFamilyNames()
```

Этот метод возвращает массив строк, содержащих имена доступных семейств шрифтов. Кроме того, в классе `GraphicsEnvironment` определен метод `getAllFonts()`, показанный ниже.

```
Font[] getAllFonts()
```

Этот метод возвращает массив объектов класса `Font`, описывающих все доступные шрифты.

Поскольку эти методы являются членами класса `GraphicsEnvironment`, для их вызова вам понадобится ссылка на объект этого класса. Вы можете получить эту ссылку, используя статический метод `getLocalGraphicsEnvironment()`, который определен в классе `GraphicsEnvironment`.

```
static GraphicsEnvironment getLocalGraphicsEnvironment()
```

Рассмотрим пример аплета, который показывает, как получить имена всех доступных семейств шрифтов.

```
// Отображение шрифтов.
/*
<applet code="ShowFonts" width=550 height=60>
</applet>
*/

import java.applet.*;
import java.awt.*;

public class ShowFonts extends Applet {
    public void paint(Graphics g) {
        String msg = "";
        String FontList[];

        GraphicsEnvironment ge =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
        FontList = ge.getAvailableFontFamilyNames();
        for(int i = 0; i < FontList.length; i++)
            msg += FontList[i] + " ";
        g.drawString(msg, 4, 16);
    }
}
```

Примерный вывод этого аплета показан на рис. 24.11. Однако список шрифтов у вас может отличаться от показанного на рисунке.

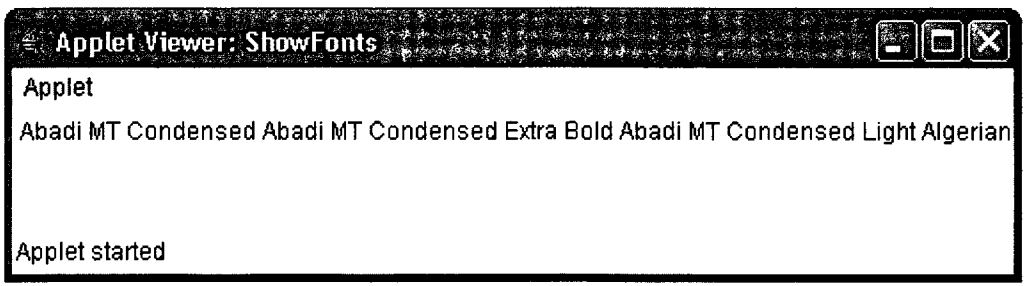


Рис. 24.11. Пример получения списка доступных шрифтов

Создание и выбор шрифта

Чтобы выбрать новый шрифт, следует сначала создать объект класса `Font`, описывающий шрифт. Один конструктор класса `Font` имеет следующую общую форму.

`Font(String` *`имяШрифта`*`, int` *`стильШрифта`*`, int` *`размерВпунктах`*`)`

Здесь *`имяШрифта`* определяет имя желаемого шрифта. Имя может быть задано с использованием либо логического имени, либо названия гарнитуры. Все среды Java поддерживают следующие шрифты: Dialog, DialogInput, Sans Serif и Monospaced. Шрифт Dialog используется в диалоговых окнах вашей системы. Шрифт Dialog является также шрифтом по умолчанию, когда никакой другой шрифт не установлен явно. Также вы можете использовать любой другой шрифт, который поддерживает ваша конкретная среда, однако будьте осторожны — эти шрифты могут быть не всегда доступны.

Стиль шрифта указан параметром *`стильШрифта`*. Он может состоять из одной или более констант: `Font.PLAIN`, `Font.BOLD` и `Font.ITALIC`. Чтобы скомбинировать стили, объединяйте их с помощью оператора "ИЛИ". Например, `Font.BOLD` и `Font.ITALIC` задают полужирный курсив.

Размер шрифта в точках указывается параметром *размерВпунктах*.

Чтобы использовать только что созданный шрифт, следует выбрать его с помощью метода `setFont()`, определенного в классе `Component`. Вот его общая форма.

```
void setFont(Font объектШрифта)
```

Здесь *объектШрифта* — это объект, содержащий желаемый шрифт.

Следующая программа выводит примеры каждого стандартного шрифта. Всякий раз, когда вы щелкаете кнопкой мыши внутри окна, выбирается новый шрифт и отображается его имя.

```
// Отображение шрифтов.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/*
<applet code="SampleFonts" width=200 height=100>
</applet>
*/

public class SampleFonts extends Applet {
    int next = 0;
    Font f;
    String msg;

    public void init() {
        f = new Font("Dialog", Font.PLAIN, 12);
        msg = "Dialog";
        setFont(f);
        addMouseListener(new MyMouseAdapter(this));
    }

    public void paint(Graphics g) {
        g.drawString(msg, 4, 20);
    }
}

class MyMouseAdapter extends MouseAdapter {
    SampleFonts sampleFonts;
    public MyMouseAdapter(SampleFonts sampleFonts) {
        this.sampleFonts = sampleFonts;
    }
    public void mousePressed(MouseEvent me) {
        // Переключать шрифты с каждым щелчком 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 этого аплета показан на рис. 24.12.

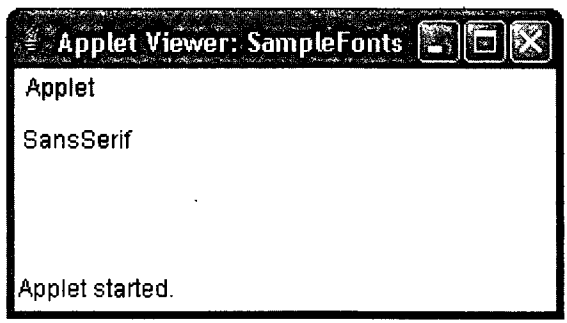


Рис. 24.12. Пример отображения шрифтов

Получение информации о шрифте

Предположим, что вы хотите получить информацию о текущем выбранном шрифте. Для этого следует сначала получить текущий шрифт вызовом метода `getFont()`. Этот метод определен в классе `Graphics`, как показано ниже.

```
Font getFont()
```

Получив текущий выбранный шрифт, вы можете извлечь информацию о нем с использованием различных методов, определенных в классе `Font`. Например, следующий аплет отображает имя, семейство, размер и стиль текущего выбранного шрифта.

```
// Отображение информации о шрифте.
import java.applet.*;
import java.awt.*;

/*
<applet code="FontInfo" width=350 height=60>
</applet>
*/

public class FontInfo extends Applet {
    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;
     // String msg = "Семейство: " + fontName;

        msg += ", Font: " + fontFamily;
```

```
      // msg += ", Шрифт: " + fontFamily;

         msg += ", Size: " + fontSize + ", Style: ";
      // msg += ", Размер: " + fontSize + ", Style: ";

         if((fontStyle & Font.BOLD) == Font.BOLD)
             msg += "Bold ";
          // msg += "Полужирный ";

         if((fontStyle & Font.ITALIC) == Font.ITALIC)
             msg += "Italic ";
          // msg += "Курсив ";

         if((fontStyle & Font.PLAIN) == Font.PLAIN)
             msg += "Plain ";
          // msg += "Обычный ";
      g.drawString(msg, 4, 16);
      }
}
```

Управление выводом текста с использованием класса `FontMetrics`

Как только что было сказано, язык Java поддерживает множество шрифтов. Для большинства из них символы не имеют одинакового размера — большинство шрифтов пропорциональны. Высота каждого символа, длина *подстрочных элементов*, таких как нижняя часть некоторых букв, например *у*, а также промежуток между горизонтальными линиями варьируются от шрифта к шрифту. Более того, размер шрифта в точках также может изменяться. Все эти (и другие) атрибуты являются переменными и не представляют особого интереса для вас, как программиста, поскольку язык Java не требует ручного управления почти всем текстовым выводом.

Учитывая, что размер каждого шрифта может отличаться и что шрифты могут изменяться в процессе выполнения вашей программы, должен существовать какой-то способ определения размеров и прочих разнообразных атрибутов текущего выбранного шрифта. Например, чтобы вывести одну строку текста после другой, необходимо каким-то образом узнать высоту шрифта и количество пикселей между строками. Чтобы позволить это, библиотека AWT предусматривает класс `FontMetrics`, инкапсулирующий разнообразную информацию о шрифте. Начнем с определения общей терминологии, используемой при описании шрифтов (табл. 24.4).

Таблица 24.4. Общая терминология, используемая при описании шрифтов

Высота (height)	Размер строки текста сверху вниз
Базовая линия (baseline)	Строка, по которой выровнены нижние грани символов (за исключением подстрочных элементов)
Возвышение (ascent)	Расстояние от базовой линии до вершины символов
Спуск (descent)	Расстояние от базовой линии до нижней точки символов
Междустрочный интервал (leading)	Расстояние между нижней точкой строки текста и ее верхней точкой

Как вы, наверняка, заметили, метод `drawString()` использовался во многих предыдущих примерах. Он выводит строку текущего цвета текущим шрифтом, на-

чиная с указанного местоположения. Однако это местоположение находится на левой грани базовой линии символов, а не в верхней левой точке, как это принято в других методах рисования. Часто допускается ошибка, заключающаяся в попытке рисовать строку по тем же координатам, где вы рисовали бы рамку. Например, если вам нужно нарисовать прямоугольник начиная с координаты (0,0), то вы увидите полный прямоугольник. Если же вы попытаетесь вывести строку "Typesetting" начиная с координаты (0,0), то увидите только подстрочные элементы букв *y*, *p* и *g*. Как вы убедитесь далее, используя метрики шрифта, можно определить правильное местоположение каждой строки, подлежащей отображению.

Класс FontMetrics определяет несколько методов, которые помогают вам управлять текстовым выводом. Несколько наиболее часто используемых методов перечислено в табл. 24.5. Эти методы помогают правильно отобразить текст в окне. Рассмотрим некоторые из них на примерах.

Таблица 24.5. Выборка методов, определенных в классе FontMetrics

Метод	Описание
`int bytesWidth(byte b[], int начало, int колБайтов)`	Возвращает ширину *колБайтов* символов, содержащихся в массиве *b*, начиная с *начало*
`int charWidth(char c[], int начало, int количСимволов)`	Возвращает ширину *количСимволов* символов, содержащихся в массиве *c*, начиная с *начало*
`int charWidth(char c)`	Возвращает ширину *c*
`int charWidth(int c)`	Возвращает ширину *c*
`int getAscent()`	Возвращает возвышение шрифта
`int getDescent()`	Возвращает спуск шрифта
`Font getFont()`	Возвращает текущий шрифт
`int getHeight()`	Возвращает высоту строки текста. Это значение может быть использовано для вывода множества строк текста в окно
`int getLeading()`	Возвращает пробел между строками текста
`int getMaxAdvance()`	Возвращает ширину наиболее широкого символа. Если это значение недоступно, возвращается значение -1
`int getMaxAscent()`	Возвращает максимальное возвышение
`int getMaxDescent()`	Возвращает максимальный спуск
`int[] getWidths()`	Возвращает ширину первых 256 символов
`int stringWidth(String строка)`	Возвращает ширину строки, указанной в *строка*
`String toString()`	Возвращает строковый эквивалент вызывающего объекта

Отображение множества строк текста

Возможно, чаще всего класс FontMetrics используется для определения расстояния между строками текста. Кроме того, он определяет длину строки, которую нужно отобразить. Ниже вы увидите, как решаются эти задачи.

Для того чтобы отобразить многострочный текст, ваша программа должна отслеживать текущую позицию вывода. Всякий раз, когда встречается символ новой строки, координата Y должна быть перенесена в начало следующей строки. Когда

отображается строка, координата X должна быть установлена в точку, где эта строка заканчивается. Это позволит начать вывод следующей строки так, чтобы она начиналась сразу после окончания предыдущей.

Чтобы определить расстояние между строками, вы можете использовать значение, возвращенное методом `getLeading()`. Чтобы определить общую высоту шрифта, добавьте значение, возвращенное методом `getAscent()`, к значению, возвращенному методом `getDescent()`. Затем вы можете использовать эти значения для позиционирования каждой строки текста, подлежащего выводу. Однако во многих случаях вам не понадобятся эти значения по отдельности. Зачастую вам нужно знать лишь общую высоту строки, представляющую сумму междустрочного интервала и значений подъема и спуска шрифта. Самый простой путь получения этого значения — вызвать метод `getHeight()`. Просто увеличивайте координату Y на эту величину всякий раз, когда хотите перейти на следующую строку выводимого текста.

Чтобы начать вывод с конца предыдущего вывода в той же строке, следует знать длину в пикселях каждой строки, которую отображаете. Чтобы получить это значение, вызовите метод `stringWidth()`. Вы можете использовать это значение для вычисления координаты X всякий раз, когда выводите строку.

В следующем аплете показано, как вывести многострочный текст в окно. Он также отображает несколько предложений на одной и той же строке. Обратите внимание на переменные `curX` и `curY`. Они отслеживают текущую позицию вывода текста.

```
// Демонстрация многострочного вывода.
import java.applet.*;
import java.awt.*;
/*
<applet code="MultiLine" width=300 height=100>
</applet>
*/

public class MultiLine extends Applet {
    int curX=0, curY=0; // текущая позиция
    public void init() {
        Font f = new Font("SansSerif", Font.PLAIN, 12);
        setFont(f);
    }

    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 = curY = 0; // Сбросить координаты для каждой перерисовки.

     // nextLine("Это в строке один.", g);
     // nextLine("Это в строке два.", g);
     // sameLine(" Это в той же строке.", g);
     // sameLine(" И это тоже.", g);
     // nextLine("Это в строке три.", g);
    }

    // Переход на следующую строку.
    void nextLine(String s, Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        curY += fm.getHeight(); // переход на следующую строку
        curX = 0;
        g.drawString(s, curX, curY);
        curX = fm.stringWidth(s); // переход к концу строки
```

```
    }

    // Отображение в той же строке.
    void sameLine(String s, Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        g.drawString(s, curX, curY);
        curX += fm.stringWidth(s); // переход к концу строки
    }
}
```

Примерный вывод этого аплета показан на рис. 24.13.

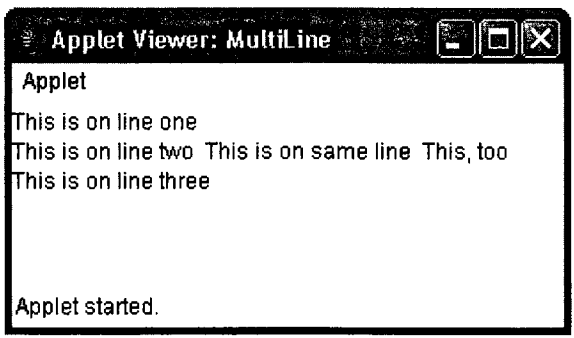


Рис. 24.13. Пример многострочного вывода

Центрирование текста

Рассмотрим пример, центрирующий текст в окне слева направо и сверху вниз. Он получает возвышение, спуск и ширину строки и вычисляет позицию, в которой он должен быть отображен для центрирования.

```
// Центрирование текста.
import java.applet.*;
import java.awt.*;

/*
<applet code="CenterText" width=200 height=100>
</applet>
*/

public class CenterText extends Applet {
    final Font f = new Font("SansSerif", Font.BOLD, 18);

    public void paint(Graphics g) {
        Dimension d = this.getSize();

        g.setColor(Color.white);
        g.fillRect(0, 0, d.width,d.height);
        g.setColor(Color.black);
        g.setFont(f);
        drawCenteredString("This is centered.", d.width, d.height, g);
     // drawCenteredString("Центрировано.", d.width, d.height, g);
        g.drawRect(0, 0, d.width-1, d.height-1);
    }

    public void drawCenteredString(String s, int w, int h, Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        int x = (w - fm.stringWidth(s)) / 2;
        int y = (fm.getAscent() + (h - (fm.getAscent() +
```

```
                fm.getDescent()))/2);
        g.drawString(s, x, y);
    }
}
```

На рис. 24.14 показан пример вывода этого аплета.

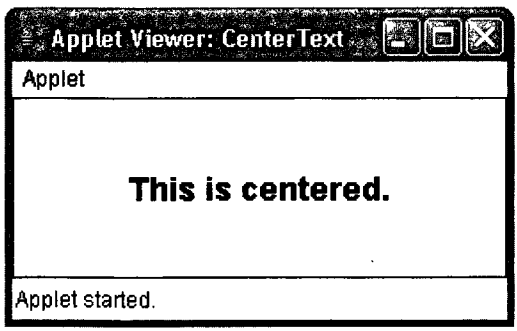


Рис. 24.14. Пример центрирования текста

Выравнивание многострочного текста

При использовании текстового процессора обычно вы видите текст выровненным таким образом, что одна или обе его стороны образуют ровную вертикальную линию. Например, большинство текстовых процессоров может выравнивать текст по левому и/или по правому краю. Большинство также может центрировать текст. В следующей программе вы увидите, как добиться этого эффекта.

В этой программе строки, подлежащие выравниванию, разбиваются на слова. Для каждого слова программа отслеживает его длину в текущем шрифте и автоматически переходит на следующую строку, когда слово не умещается в текущей. Каждая завершенная строка отображается в окне с текущим выбранным стилем выравнивания. Всякий раз, когда вы щелкаете кнопкой мыши в окне аплета, стиль выравнивания изменяется. Пример вывода этого аплета показан на рис. 24.15.

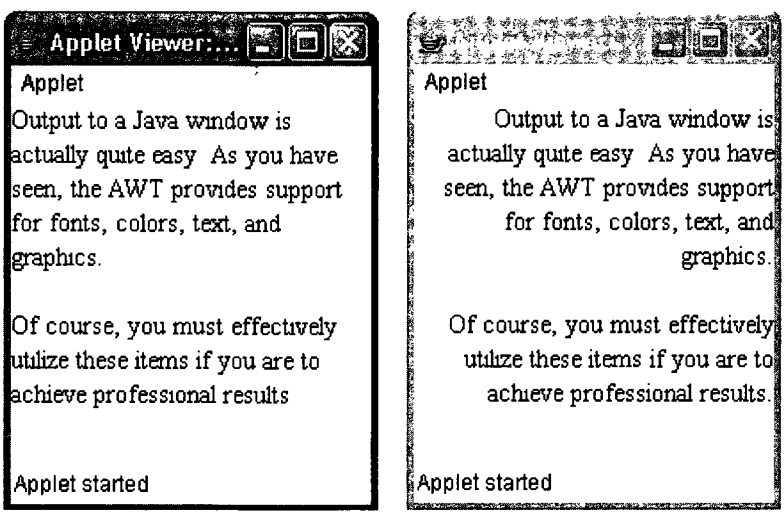


Рис. 24.15. Пример выравнивания текста

```
// Демонстрация выравнивания текста.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/* <title>Text Layout</title>
<applet code="TextLayout" width=200 height=200>
<param name="text" value="Output to a Java window is actually
    quite easy.
    As you have seen, the AWT provides support for
    fonts, colors, text, and graphics. <P> Of course,
    you must effectively utilize these items
    if you are to achieve professional results.">
    <param name="fontname" value="Serif">
    <param name="fontSize" value="14">
</applet>
*/

public class TextLayout extends Applet {
    final int LEFT = 0;
    final int RIGHT = 1;
    final int CENTER = 2;
    final int LEFTRIGHT =3;
    int align;
    Dimension d;
    Font f;
    FontMetrics fm;
    int fontSize;
    int fh, bl;
    int space;
    String text;

    public void init() {
        setBackground(Color.white);
        text = getParameter("text");
        try {
            fontSize = Integer.parseInt(getParameter("fontSize"));}
        catch (NumberFormatException e) {
            fontSize=14;
        }
        align = LEFT;
        addMouseListener(new MyMouseAdapter(this));
    }

    public void paint(Graphics g) {
    update(g);
}

public void update(Graphics g) {
    d = getSize();
    g.setColor(getBackground());
    g.fillRect(0,0,d.width, d.height);
    if(f==null) f = new Font(getParameter("fontname"),
                        Font.PLAIN, fontSize);
    g.setFont(f);

    if(fm == null) {
        fm = g.getFontMetrics();
        bl = fm.getAscent();
        fh = bl + fm.getDescent();
```

```
            space = fm.stringWidth(" ");
        }

        g.setColor(Color.black);
        StringTokenizer st = new StringTokenizer(text);
        int x = 0;
        int nextx;
        int y = 0;
        String word, sp;
        int wordCount = 0;
        String line = "";
        while (st.hasMoreTokens()) {
            word = st.nextToken();
            if(word.equals("<P>")) {
                drawString(g, line, wordCount,
                           fm.stringWidth(line), y+bl);
                line = "";
                wordCount = 0;
                x = 0;
                y = y + (fh * 2);
            }
            else {
                int w = fm.stringWidth(word);
                if(( nextx = (x+space+w)) > d.width ) {
                    drawString(g, line, wordCount,
                               fm.stringWidth(line), y+bl);
                    line = "";
                    wordCount = 0;
                    x = 0;
                    y = y + fh;
                }
                if(x!=0) {sp = " ";} else {sp = "";}
                line = line + sp + word;
                x = x + space + w;
                wordCount++;
            }
        }
        drawString(g, line, wordCount, fm.stringWidth(line), y+bl);
    }

    public void drawString(Graphics g, String line,
                           int wc, int lineW, int y) {
        switch(align) {
            case LEFT: g.drawString(line, 0, y);
                break;
            case RIGHT: g.drawString(line, d.width-lineW ,y);
                break;
            case CENTER: g.drawString(line, (d.width-lineW)/2, y);
                break;
            case LEFTRIGHT:
                if(lineW < (int)(d.width*.75)) {
                    g.drawString(line, 0, y);
                }
                else {
                    int toFill = (d.width - lineW)/wc;
                    int nudge = d.width - lineW - (toFill*wc);
                    int s = fm.stringWidth(" ");
                    StringTokenizer st = new StringTokenizer(line);
                    int x = 0;
                    while(st.hasMoreTokens()) {
                        String word = st.nextToken();
```

```
                    g.drawString(word, x, y);
                    if(nudge>0) {
                        x = x + fm.stringWidth(word) + space + toFill +
                            toFill + 1;
                        nudge--;
                    } else {
                        x = x + fm.stringWidth(word) + space + toFill;
                    }
                }
            }
            break;
        }
    }
}
class MyMouseAdapter extends MouseAdapter {
    TextLayout tl;
    public MyMouseAdapter(TextLayout tl) {
        this.tl = tl;
    }

    public void mouseClicked(MouseEvent me) {
        tl.align = (tl.align + 1) % 4;
        tl.repaint();
    }
}
```

Давайте разберемся, как работает этот аплет. Сначала в нем создается несколько констант, которые будут использоваться для определения стиля выравнивания, затем объявляется несколько переменных. Метод `init()` получает текст, подлежащий отображению. Далее он инициализирует размер шрифта в блоке `try-catch`, который затем установит размер шрифта равным 14, если параметр `fontSize` опущен в дескрипторе HTML. Параметр `text` представляет собой длинную строку текста с дескриптором HTML `<P>` в качестве разделителя абзаца.

Метод `update()` — "двигатель" нашего примера. Он устанавливает шрифт и получает базовую линию и высоту из объекта метрик шрифта. Далее он создает объект класса `StringTokenizer` и использует его для извлечения следующей лексемы (строки, отделенной пробелами) из строки, заданной в параметре `text`. Если следующей лексемой является дескриптор `<P>`, осуществляется вертикальная прокрутка. В противном случае метод `update()` проверяет, вписывается ли длина лексемы в текущем шрифте в ширину колонки. Если строка уже заполнена текстом или лексем для вывода более не осталось, строка выводится специальной версией метода `drawString()`. Первые три случая вызова метода `drawString()` просты. Каждый выравнивает строку, переданную переменной `line`, по левому или правому краю или же по центру колонки, в зависимости от текущего стиля выравнивания. В случае стиля `LEFTRIGHT` выравниваются обе стороны строки. Это значит, что необходимо вычислить оставшееся пространство (как разницу между шириной строки и шириной колонки) и распределить его между словами. Последний метод класса переключает стиль выравнивания при каждом щелчке кнопкой мыши в окне аплета.

ГЛАВА 25

Использование элементов управления, диспетчеров компоновки и меню библиотеки AWT

В этой главе продолжим исследование библиотеки AWT. Сначала рассмотрим стандартные элементы управления и диспетчеры компоновки. Затем речь пойдет о меню и полосе меню. Мы поговорим о двух компонентах верхнего уровня — диалоговом окне и диалоговом окне выбора файла. В конце главы предложим другой взгляд на вопрос обработки событий.

Элементами управления называются компоненты, которые позволяют пользователю взаимодействовать с вашим приложением различными способами; например, наиболее распространенным элементом управления является экранная кнопка. *Диспетчер компоновки* автоматически позиционирует компоненты внутри контейнера. Поэтому внешний вид окна зависит как от элементов управления, которые оно содержит, так и от диспетчера компоновки, используемого для их позиционирования.

Кроме элементов управления, обрамляющее окно может также включать *полосу меню* стандартного стиля. Каждый пункт полосы меню раскрывает меню, в котором пользователь может выбрать необходимую ему команду. Полоса меню всегда располагается в верхней части окна.

Несмотря на различие во внешнем виде, полосы меню обрабатываются почти так же, как и другие элементы управления. Несмотря на то что позиционировать компоненты в окне можно вручную, сделать это обычно непросто. Диспетчер компоновки выполняет эту задачу автоматически. В начале этой главы, где рассматриваются различные элементы управления, используется диспетчер компоновки, принятый по умолчанию. Он отображает компоненты в контейнере, размещая их слева направо и сверху вниз. После того как рассмотрим элементы управления, поговорим о диспетчерах компоновки. Вы узнаете, как лучше всего управлять позиционированием элементов управления.

Основы элементов управления

Библиотека AWT поддерживает следующие типы элементов управления:

- метки;
- экранные кнопки;
- флажки;
- списки выбора;
- списки;
- полосы прокрутки;
- элементы редактирования текста.

Все эти элементы управления относятся к подклассам класса `Component`.

Добавление и удаление элементов управления

Чтобы включить элемент управления в окно, его нужно сначала добавить к нему. Для этого необходимо создать экземпляр требуемого элемента управления, после чего добавить его в окно с помощью метода `add()`, который определен классом `Container`. Метод `add()` имеет несколько форм. В начале этой главы используется следующая форма.

```
Component add(Component объектУпр)
```

Здесь параметр *объектУпр* представляет экземпляр элемента управления, который вы хотите добавить. Метод возвращает ссылку на объект *объектУпр*. После того как элемент управления будет добавлен, он будет отображаться при появлении его родительского окна.

Иногда необходимо удалить элементы управления из окна. Для этого служит метод `remove()`. Он тоже определен в классе `Container`. Вот одна из его форм.

```
void remove(Component объект)
```

Здесь параметр *объект* представляет ссылку на элемент управления, который вы хотите удалить. Можно удалить все элементы управления, если вызвать метод `removeAll()`.

Реакция на действия над элементами управления

За исключением меток, которые являются пассивными элементами, каждый элемент управления извещает о событии в тот момент, когда к нему обращается пользователь. Например, когда пользователь щелкает на экранной кнопке, происходит событие, идентифицирующее эту кнопку. В общем случае ваша программа просто реализует соответствующий интерфейс, после чего регистрирует слушателя события для каждого элемента управления, за которым вы хотите вести наблюдение. Как было сказано в главе 23, если установить слушатель, то извещения о событиях будут передаваться ему автоматически. В последующих разделах для каждого элемента управления будет описан соответствующий интерфейс.

Исключение `HeadlessException`

Большинство элементов управления библиотеки AWT, рассматриваемых в этой главе, имеют конструкторы, способные передавать исключение `HeadlessException` при попытке создать экземпляр компонента GUI в неинтерактивной среде (т.е. в среде, в которой, например, нет монитора, мыши или клавиатуры). Исключение `HeadlessException` было добавлено в Java 1.4. С помощью этого исключения можно написать код, который будет адаптироваться к неинтерактивным средам. (Естественно, делать это можно не всегда.) Данное исключение не обрабатывается программами в этой главе, так как для иллюстрации элементов управления библиотеки AWT нужна интерактивная среда.

Метки

Самым простым элементом управления является метка. *Метка* представляет собой объект класса `Label` и содержит строку, которую она отображает. Метки являются пассивными элементами управления, не поддерживающими никакого взаимодействия с пользователем. Класс `Label` определяет следующие конструкторы.

```
Label() throws HeadlessException
Label(String строка) throws HeadlessException
Label(String строка, int как) throws HeadlessException
```

В первом варианте создается пустая метка, во втором — метка, содержащая строку *строка*; эта строка выравнивается по левому краю. В третьем варианте создается метка, содержащая строку *строка*, выравнивание которой определяется с помощью параметра *как*. Параметр *как* должен принимать одну из следующих трех констант: `Label.LEFT`, `Label.RIGHT` или `Label.CENTER`.

Изменить состояние текста в метке можно с помощью метода `setText()`. Получить текущую метку можно с помощью метода `getText()`. Эти методы показаны далее.

```
void setText(String строка)
String getText()
```

В методе `setText()` параметр *строка* определяет новую метку. Метод `getText()` возвращает текущую метку.

Выровнять строку в метке можно с помощью метода `setAlignment()`. Чтобы получить текущее выравнивание, используется метод `getAlignment()`. Эти методы показаны ниже.

```
void setAlignment(int как)
int getAlignment()
```

Здесь параметр *как* должен представлять одну из трех приведенных выше констант. В следующем примере создаются три метки, которые затем добавляются в аплет.

```
// Демонстрация меток
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/

public class LabelDemo extends Applet {
    public void init() {
        Label one = new Label("One");
        Label two = new Label("Two");
        Label three = new Label("Three");

        // добавление меток в окно аплета
        add(one);
        add(two);
        add(three);
    }
}
```

На рис. 25.1 показано окно, созданное аплетом `LabelDemo`. Обратите внимание на то, что упорядочение меток в окне выполнено при помощи диспетчера компоновки, используемого по умолчанию. Позже вы увидите, каким образом можно более точно управлять размещением меток.

Использование кнопок

Пожалуй, наиболее широко используемым элементом управления является кнопка. *Экранная кнопка* — это компонент, который содержит метку и извещает о событии, когда пользователь щелкает на ней кнопкой мыши. Экранные кнопки являются объектами класса `Button`. Класс `Button` определяет следующие конструкторы.

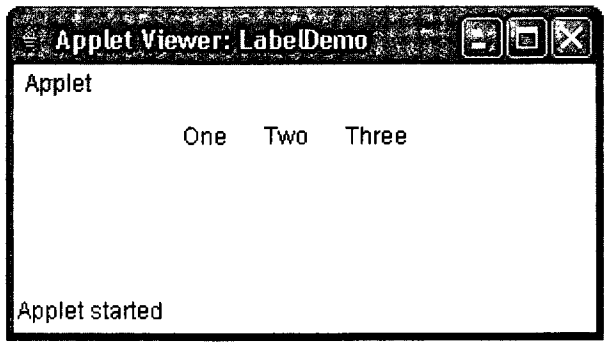


Рис. 25.1. Окно аплета `LabelDemo`

```
Button() throws HeadlessException
Button(String строка) throws HeadlessException
```

В первом варианте создается пустая кнопка, во втором — кнопка с меткой *строка*.

После того как кнопка создана, с помощью метода `setLabel()` можно определить ее метку. Получить метку можно с помощью метода `getLabel()`. Эти методы показаны ниже.

```
void setLabel(String строка)
String getLabel()
```

Здесь параметр *строка* представляет новую метку для кнопки.

Обработка кнопок

Когда пользователь щелкает на кнопке, происходит событие действия. Извещение о нем посылается любому слушателю, который предварительно зарегистрировался на получение извещений о событиях действия от данного компонента. Каждый слушатель реализует интерфейс `ActionListener`. Этот интерфейс определяет метод `actionPerformed()`, который вызывается во время события. В качестве параметра для этого метода указывается объект класса `ActionEvent`. Он содержит ссылку на кнопку, извещающую о событии, и ссылку на *командную строку действия*, связанную с этой кнопкой. По умолчанию командной строкой действия является метка кнопки. Как правило, для идентификации кнопки используется или ссылка на кнопку, или командная строка действия. (Скоро вы увидите примеры каждого подхода.)

Ниже показан пример создания трех кнопок с метками `"Yes"` (Да), `"No"` (Нет) и `"Undecided"` (Нет решения). Когда пользователь щелкает на одной из кнопок, отображается сообщение, свидетельствующее о выбранной кнопке. В этом варианте команда действия кнопки (которая по умолчанию является ее меткой) используется для определения, на какой из кнопок выполнен щелчок. Получение метки производится при помощи вызова метода `getActionCommand()` в объекте класса `ActionEvent`, который передается методу `actionPerformed()`.

```
/// Пример кнопок
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/

public class ButtonDemo extends Applet implements ActionListener {
```

```
    String msg = "";
    Button yes, no, maybe;

    public void init() {
        yes = new Button("Yes");
        no = new Button("No");
        maybe = new Button("Undecided");

        add(yes);
        add(no);
        add(maybe);

        yes.addActionListener(this);
        no.addActionListener(this);
        maybe.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae) {
        String str = ae.getActionCommand();
        if(str.equals("Yes")) {
            msg = "You pressed Yes.";
         // msg = "Нажата кнопка Yes.";
        }
        else if(str.equals("No")) {
            msg = "You pressed No.";
         // msg = "Нажата кнопка No.";
        }
        else {
            msg = "You pressed Undecided.";
         // msg = "Нажата кнопка Undecided.";
        }
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString(msg, 6, 100);
    }
}
```

Окно функционирующего аплета `ButtonDemo` можно видеть на рис. 25.2.

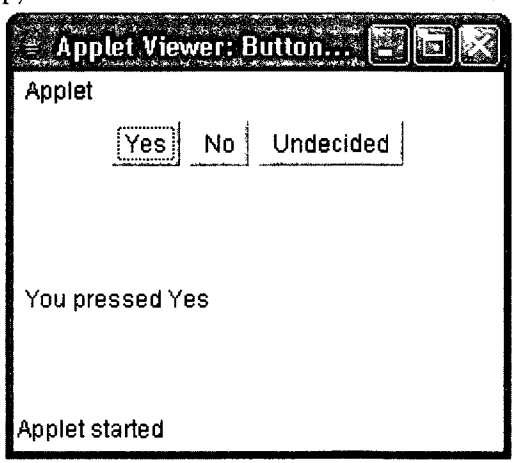


Рис. 25.2. Окно аплета `ButtonDemo`

Как уже было сказано, помимо сравнения командных строк действия кнопок, можно также определить, на какой из кнопок выполнен щелчок, если сравнить объект, полученный из метода `getSource()`, с объектами кнопок, которые вы добавляете в окно. Для этого следует вести список добавляемых объектов. Этот подход отражен в следующем аплете.

```
// Распознавание объектов Button.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonList" width=250 height=150>
</applet>
*/

public class ButtonList extends Applet implements ActionListener {
    String msg = "";
    Button bList[] = new Button[3];

    public void init() {
        Button yes = new Button("Yes");
        Button no = new Button("No");
        Button maybe = new Button("Undecided")

        // сохранение ссылок на кнопки при добавлении
        bList[0] = (Button) add(yes);
        bList[1] = (Button) add(no);
        bList[2] = (Button) add(maybe);

        // регистрация на получение уведомлений о событиях действия
        for(int i = 0; i < 3; i++) {
            bList[i].addActionListener(this);
        }
    }

    public void actionPerformed(ActionEvent ae) {
        for(int i = 0; i < 3; i++) {
            if(ae.getSource() == bList[i]) {
                msg = "You pressed " + bList[i].getLabel();
             // msg = "Нажата кнопка " + bList[i].getLabel();
            }
        }
        repaint();
    }
    public void paint(Graphics g) {
        g.drawString(msg, 6, 100);
    }
}
```

В этом варианте при добавлении кнопок в окно аплета программа записывает в массив ссылку на каждую кнопку. (Вспомните, что метод `add()` возвращает ссылку на кнопку при ее добавлении.) Впоследствии этот массив используется в методе `actionPerformed()`, чтобы определить, на какой кнопке выполнен щелчок.

В простых аплетах распознать кнопки по их меткам обычно несложно. Но если вы будете изменять метку внутри кнопки во время выполнения или использовать кнопки с одинаковыми метками, то определить, на какой кнопке выполнен щелчок, можно очень легко, если воспользоваться ее объектной ссылкой. Можно также присвоить строке команды действия, связанной с кнопкой, что-нибудь отличное от ее метки, вызвав метод `setActionCommand()`. Он изменяет строку команды действия, не влияя

на строку, используемую в качестве метки кнопки. Таким образом, если задать строку команды действия, можно отделить друг от друга команду действия и метку кнопки.

Использование флажков

Флажок представляет собой элемент управления, который служит для включения или отключения чего-нибудь. Он состоит из небольшого окна, которое может либо содержать отметку (в виде "галочки"), либо быть пустым. У каждого флажка есть метка, описывающая параметр, который представляет флажок. При щелчке на флажке можно изменить его состояние. Флажки могут использоваться как индивидуально, так и в составе группы. Они являются объектами класса `Checkbox`.

Класс `Checkbox` поддерживает следующие конструкторы.

```
Checkbox() throws HeadlessException
Checkbox(String строка) throws HeadlessException
Checkbox(String строка, boolean вкл) throws HeadlessException
Checkbox(String строка, boolean вкл, CheckboxGroup группаФлажка) throws
HeadlessException
Checkbox(String строка, CheckboxGroup группаФлажка, boolean вкл) throws
HeadlessException
```

Первая форма конструктора создает флажок, метка которого изначально является пустой. Флажок находится в сброшенном состоянии. Вторая форма создает флажок, метка которого определяется параметром *строка*. Флажок находится в сброшенном состоянии. Третья форма позволяет установить исходное состояние флажка. Если параметр *вкл* содержит значение `true`, то флажок изначально будет установлен, в противном случае — нет. Четвертая и пятая формы создают флажок, метка которого определяется параметром *строка*, а группа — параметром *группаФлажка*. Если флажок не является частью группы, то параметр *группаФлажка* должен иметь значение `null`. (Группы флажков описываются в следующем разделе.) Значение параметра *вкл* определяет исходное состояние флажка.

Чтобы получить текущее состояние флажка, используется метод `getState()`. Для установки его состояния используется метод `setState()`. С помощью метода `getLabel()` можно получить текущую метку, связанную с флажком. Чтобы задать метку, нужно вызвать метод `setLabel()`. Эти методы показаны ниже.

```
boolean getState()
void setState(boolean вкл)
String getLabel()
void setLabel(String строка)
```

Если параметр *вкл* содержит значение `true`, то флажок будет установлен. Если он содержит значение `false`, флажок будет сброшен. Строка, передаваемая в параметре *строка*, становится новой меткой, связанной с вызываемым флажком.

Обработка флажков

Каждый раз, когда флажок устанавливается или сбрасывается, происходит событие элемента. Извещение о нем передается любому слушателю, который ранее зарегистрировался на получение извещений о событиях элемента от данного компонента. Каждый слушатель реализует интерфейс `ItemListener`. Этот интерфейс определяет метод `itemStateChanged()`. Объект класса `ItemEvent` задается в качестве параметра для данного метода. Он содержит информацию о событии (например, выбор или отмена выбора).

В следующей программе создается четыре флажка. Первый флажок изначально установлен. Состояние каждого флажка отображается. Каждый раз при изменении состояния флажка производится обновление отображаемого состояния.

```
// Демонстрация применения флажков.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=240 height=200>
</applet>
*/

public class CheckboxDemo extends Applet implements ItemListener {
    String msg = "";
    Checkbox winXP, win7, solaris, mac;

    public void init() {
        winXP = new Checkbox("Windows XP", null, true);
        win7 = new Checkbox("Windows 7");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");
        add(winXP);
        add(win7);
        add(solaris);
        add(mac);

        winXP.addItemListener(this);
        win7.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }

    // Отображение текущего состояния флажков.
    public void paint(Graphics g) {
        msg = "Current state: ";
     // msg = "Текущее состояние: ";
        g.drawString(msg, 6, 80);
        msg = " Windows XP: " + winXP.getState();
        g.drawString(msg, 6, 100);
        msg = " Windows 7: " + win7.getState();
        g.drawString(msg, 6, 120);
        msg = " Solaris: " + solaris.getState();
        g.drawString(msg, 6, 140);
        msg = " Mac OS: " + mac.getState();
        g.drawString(msg, 6, 160);
    }
}
```

Окно функционирующего аплета показано на рис. 25.3.

Класс CheckboxGroup

Допускается создание набора взаимоисключающих флажков, в котором в одно и то же время может быть установлен один (и только один) флажок. Эти флажки часто называются *переключателями*, потому что подобны переключателю каналов в ав-

томобильном радиоприемнике — в одно и то же время можно выбрать только одну радиостанцию. Чтобы создать набор взаимоисключающих кнопок, следует вначале определить группу, к которой они принадлежат, а затем указать эту группу при создании флажков. Группы флажков представляют собой объекты класса `CheckboxGroup`. Определен только конструктор по умолчанию, создающий пустую группу.

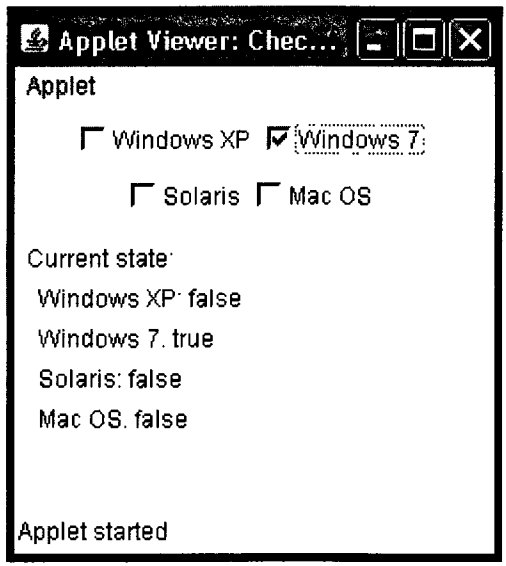


Рис. 25.3. Окно аплета `CheckboxDemo`

Чтобы узнать, какой флажок в группе установлен на данный момент, вызовите метод `getSelectedCheckBox()`. Установить флажок можно с помощью метода `setSelectedCheckbox()`. Эти методы показаны далее.

```
Checkbox getSelectedCheckbox()
void setSelectedCheckbox(Checkbox которая)
```

Здесь параметр *которая* представляет флажок, который вы хотите установить. Флажок, установленный ранее, будет сброшен. Ниже показан пример программы, в которой используются флажки, являющиеся частью группы.

```
// Демонстрация применения группы флажков.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CBGroup" width=254 height=200>
</applet>
*/

public class CBGroup extends Applet implements ItemListener {
    String msg = "";
    Checkbox winXP, win7, solaris, mac;
    CheckboxGroup cbg;

    public void init() {
        cbg = new CheckboxGroup();
        winXP = new Checkbox("Windows XP", cbg, true);
        win7 = new Checkbox("Windows 7", cbg, false);
```

```
        solaris = new Checkbox("Solaris", cbg, false);
        mac = new Checkbox("Mac OS", cbg, false);

        add(winXP);
        add(win7);
        add(solaris);
        add(mac);

        winXP.addItemListener(this);
        win7.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }

    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }

    // Отображение текущего состояния флажков.
    public void paint(Graphics g) {
        msg = "Current selection: ";
     // msg = "Текущий выбор: ";
        msg += cbg.getSelectedCheckbox().getLabel();
        g.drawString(msg, 6, 100);
    }
}
```

Окно аплета `CBGroup` во время выполнения показано на рис. 25.4. Обратите внимание на то, что в данном случае флажки имеют форму окружности.

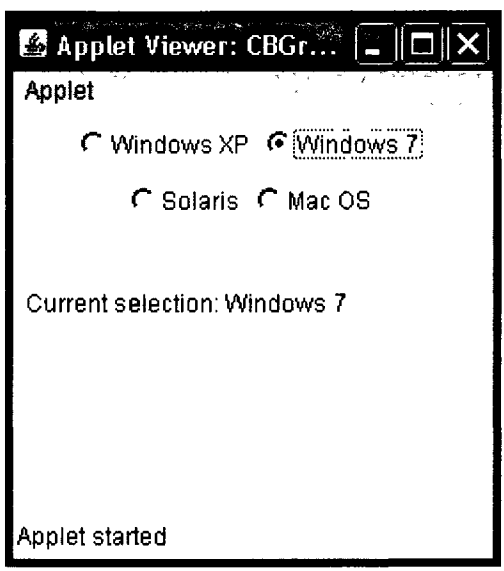


Рис. 25.4. Окно аплета `CBGroup`

Элементы управления выбором

Класс `Choice` используется для создания всплывающего списка элементов, в котором пользователь может делать свой выбор. Поэтому элемент управления `Choice`

является разновидностью меню. Будучи неактивным, компонент `Choice` занимает ровно столько пространства, сколько необходимо для отображения выбранного на данный момент элемента. Когда пользователь щелкает на нем, раскрывается весь список, в котором можно выбрать новый элемент. Каждый элемент в списке представляет собой строку, которая появляется в виде метки с выравниванием влево в том порядке, в каком он добавляется в объект класса `Choice`. Класс `Choice` определяет только стандартный конструктор, который создает пустой список.

Чтобы добавить элемент выбора в список, нужно вызвать метод `add()`. Он имеет следующую общую форму.

```
void add(String имя)
```

Здесь параметр *имя* представляет добавляемый элемент. Добавление элементов в список производится в том порядке, в каком выполняются вызовы метода `add()`.

Чтобы узнать, какой элемент выбран на данный момент, можно вызвать один из двух методов: `getSelectedItem()` или `getSelectedIndex()`.

```
String getSelectedItem()
int getSelectedIndex()
```

Метод `getSelectedItem()` возвращает строку, содержащую имя элемента, а метод `getSelectedIndex()` — индекс элемента. Первый элемент имеет индекс `0`. По умолчанию выбран первый элемент, добавленный в список.

Чтобы узнать, сколько элементов содержится в списке, нужно вызвать метод `getItemCount()`. Чтобы указать элемент, выбранный на данный момент, можно передать методу `select()` начинающийся с нуля целочисленный индекс или строку, совпадающую с именем элемента в списке. Эти методы показаны ниже.

```
int getItemCount()
void select(int индекс)
void select(String имя)
```

Для заданного индекса можно получить имя, связанное с элементом в этом индексе, если вызвать метод `getItem()`, который имеет следующую общую форму.

```
String getItem(int индекс)
```

Здесь параметр *индекс* представляет индекс требуемого элемента.

Обработка списков выбора

Каждый раз при выборе определенного элемента происходит событие. Извещение о нем передается всем слушателям, которые предварительно зарегистрировались на получение извещений о событиях элемента от данного компонента. Каждый слушатель реализует интерфейс `ItemListener`. Этот интерфейс определяет метод `itemStateChanged()`. В качестве параметра для этого метода применяется объект класса `ItemEvent`.

Ниже показан пример создания двух меню класса `Choice`. В одном из них производится выбор операционной системы, в другом — браузера.

```
// Демонстрация применения списков выбора.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ChoiceDemo" width=300 height=180>
</applet>
*/

public class ChoiceDemo extends Applet implements ItemListener {
    Choice os, browser;
```

```
    String msg = "";

    public void init() {
        os = new Choice();
        browser = new Choice();

        // добавление элементов в список os
        os.add("Windows XP");
        os.add("Windows 7");
        os.add("Solaris");
        os.add("Mac OS");

        // добавление элементов в список browser
        browser.add("Internet Explorer");
        browser.add("Firefox");
        browser.add("Opera");

        // добавление списков выбора в окно
        add(os);
        add(browser);

        // регистрация на получение уведомлений о событиях
        os.addItemListener(this);
        browser.addItemListener(this);
    }

    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }

    // Отображение текущего выбора.
    public void paint(Graphics g) {
        msg = "Current OS: ";
     // msg = "Текущая ОС: ";
        msg += os.getSelectedItem();
        g.drawString(msg, 6, 120);
        msg = "Current Browser: ";
     // msg = "Текущий браузер: ";
        msg += browser.getSelectedItem();
        g.drawString(msg, 6, 140);
    }
}
```

Окно аплета во время выполнения показано на рис. 25.5.

Использование списков

Класс `List` предлагает компактный прокручиваемый список, в котором можно выбирать множество элементов. В отличие от объекта класса `Choice`, который отображает только один выбранный элемент меню, объект класса `List` можно создать таким образом, чтобы он показывал любое количество элементов выбора в видимом окне. Его можно создать так, чтобы он позволил выбрать несколько элементов. Класс `List` предлагает следующие конструкторы.

```
List() throws HeadlessException
List(int колСтрок) throws HeadlessException
List(int колСтрок, boolean выборНескольких) throws HeadlessException
```

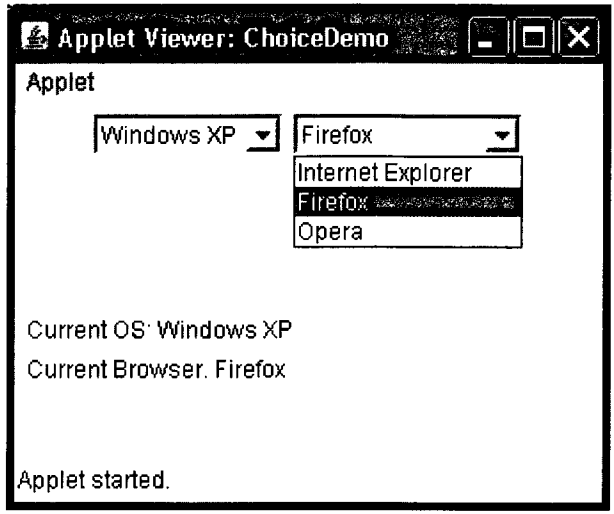


Рис. 25.5. Окно аплета `ChoiceDemo`

Первый вариант создает элемент управления класса `List`, который позволяет выбрать в одно и то же время только один элемент. Во втором варианте значение параметра *колСтрок* определяет количество пунктов в списке, которые всегда будут видимы (остальные пункты можно просмотреть, используя прокрутку). В третьем варианте, если параметр *выборНескольких* содержит значение `true`, пользователь может выбрать одновременно два или более пунктов. Если же он содержит значение `false`, то выбрать можно будет только один пункт.

Для добавления пункта в список вызовите метод `add()`. Он имеет два варианта.

```
void add(String имя)
void add(String имя, int индекс)
```

Здесь параметр *имя* определяет имя элемента, добавленного в список. В первом случае добавляются пункты в конец списка. Во втором случае добавляется пункт в индекс, определяемый параметром *индекс*. Индексация начинается с нуля. Чтобы добавить пункт в конец списка, необходимо указать значение `-1`.

Для тех списков, в которых можно выбрать только один пункт, узнать, какой из пунктов выбран на данный момент, можно с помощью метода `getSelectedItem()` или `getSelectedIndex()`. Эти методы показаны ниже.

```
String getSelectedItem()
int getSelectedIndex()
```

Метод `getSelectedItem()` возвращает строку, содержащую имя пункта. Если выбрано несколько пунктов или не был выбран еще ни один, возвращается значение `null`. Метод `getSelectedIndex()` возвращает индекс пункта. Первый пункт имеет индекс 0. Если выбрано несколько пунктов или не был выбран еще ни один, возвращается значение `-1`.

Для списков, в которых можно выбирать несколько пунктов, узнать, какие на данный момент выбраны пункты, можно с помощью методов `getSelectedItems()` или `getSelectedIndexes()`.

```
String[] getSelectedItems()
int[] getSelectedIndexes()
```

Метод `getSelectedItems()` возвращает массив, содержащий имена выбранных на данный момент пунктов. Метод `getSelectedIndexes()` возвращает массив, содержащий индексы выбранных на данный момент пунктов.

Чтобы узнать, сколько пунктов содержится в списке, вызовите метод `getItemCount()`. С помощью метода `select()` можно определить, какой пункт будет выбран на данный момент; для этих целей используется целочисленный индекс, начинающийся с нуля. Эти методы показаны ниже.

```
int getItemCount()
void select(int индекс)
```

Для данного индекса можно получить имя, связанное с пунктом в этом индексе, если вызвать метод `getItem()`, который имеет следующую общую форму.

```
String getItem(int индекс)
```

Здесь параметр *индекс* представляет индекс требуемого пункта.

Обработка списков

Для обработки событий списков необходимо реализовать интерфейс `ActionListener`. Каждый раз при двойном щелчке на пункте элемента управления класса `List` создается объект класса `ActionEvent`. Его метод `getActionCommand()` можно использовать для получения имени нового выбранного пункта. Также каждый раз при выборе или отмене выбора пункта при помощи одиночного щелчка создается объект класса `ItemEvent`. Его метод `getStateChanged()` можно использовать, чтобы узнать, чем было вызвано данное событие – выбором пункта или отменой его выбора. Метод `getItemSelectable()` возвращает ссылку на объект, инициировавший данное событие.

Ниже показан пример, где элементы управления класса `Choice` из предыдущего раздела заменены компонентами класса `List`: один – для множественного выбора, а другой – для одиночного.

```
// Демонстрация применения списков.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ListDemo" width=300 height=180>
</applet>
*/

public class ListDemo extends Applet implements ActionListener {
    List os, browser;
    String msg = "";

    public void init() {
        os = new List(4, true);
        browser = new List(4, false);
        // добавление элементов в список os
        os.add("Windows XP");
        os.add("Windows 7");
        os.add("Solaris");
        os.add("Mac OS");

        // добавление элементов в список browser
        browser.add("Internet Explorer");
        browser.add("Firefox");
        browser.add("Opera");
```

```
        browser.select(1);

        // добавление списков в окно
        add(os);
        add(browser);

        // регистрация на получение уведомлений о событиях действия
        os.addActionListener(this);
        browser.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae) {
        repaint();
    }

    // Отображение текущего выделения.
    public void paint(Graphics g) {
        int idx[];

        msg = "Current OS: ";
        idx = os.getSelectedIndexes();
        for(int i=0; i<idx.length; i++)
            msg += os.getItem(idx[i]) + " ";
        g.drawString(msg, 6, 120);
        msg = "Current Browser: ";
        msg += browser.getSelectedItem();
        g.drawString(msg, 6, 140);
    }
}
```

Аплет `ListDemo` во время выполнения показан на рис. 25.6.

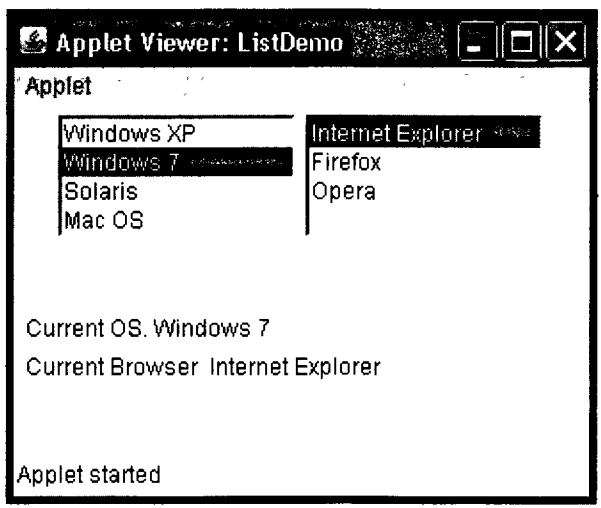


Рис. 25.6. Окно аплета `ListDemo`

Управление полосами прокрутки

Полосы прокрутки используются для выбора непрерывных значений между заданными минимумом и максимумом. Полосы прокрутки могут быть ориентированы вертикально и горизонтально. В действительности полоса прокрутки состо-

ит из нескольких отдельных частей. На обоих концах полосы находится стрелка, щелчок на которой вызывает перемещение текущего значения полосы прокрутки на одну единицу в направлении, указываемом стрелкой. Текущее значение полосы прокрутки по отношению к ее минимальным и максимальным значениям обозначается при помощи *ползунка* полосы прокрутки. Пользователь может перетащить ползунок в другую позицию, после чего полоса прокрутки отобразит новое значение. В фоновой части полосы по обе стороны ползунка пользователь может щелкнуть кнопкой мыши, чтобы осуществить переход в этом направлении с некоторым приращением больше 1. Обычно это действие переводится в некоторую разновидность перелистывания страниц вверх и вниз. Полосы прокрутки инкапсулируются классом `Scrollbar`.

Класс `Scrollbar` определяет следующие конструкторы.

```
Scrollbar()
Scrollbar(int стиль)
Scrollbar(int стиль, int исходноеЗначение, int размерПолзунка, int мин,
int макс)
```

Первый вариант создает вертикальную полосу прокрутки. Во втором и третьем вариантах можно задавать ориентацию полосы прокрутки. Если параметр *стиль* содержит значение `Scrollbar.VERTICAL`, создается вертикальная полоса прокрутки, а если значение `Scrollbar.HORIZONTAL` — горизонтальная полоса прокрутки. В третьем варианте конструктора исходное состояние полосы прокрутки определяется параметром *исходноеЗначение*. Количество единиц, представляемых высотой ползунка, задается параметром *размерПолзунка*. Минимальные и максимальные значения полосы прокрутки передаются в параметрах *мин* и *макс*.

Если вы создаете полосу прокрутки с помощью одного из первых двух конструкторов, то, прежде чем ее можно будет использовать, вам нужно определить параметры полосы прокрутки с помощью метода `setValues()`, показанного ниже.

```
void setValues(int исходноеЗначение, int размерПолзунка, int мин, int макс)
```

Параметры имеют то же значение, что и в третьем только что описанном конструкторе.

Чтобы узнать текущее значение полосы прокрутки, вызовите метод `getValue()`, который возвращает текущую настройку. Чтобы настроить текущее значение, вызовите метод `setValue()`. Эти методы показаны ниже.

```
int getValue()
void setValue(int новоеЗначение)
```

Здесь параметр *новоеЗначение* определяет новое значение для полосы прокрутки. Когда вы присваиваете значение, ползунок внутри полосы прокрутки будет расположен так, чтобы соответствовать новому значению.

Методы `getMinimum()` и `getMaximum()` позволяют получить минимальные и максимальные значения.

```
int getMinimum()
int getMaximum()
```

Методы возвращают соответствующую величину.

По умолчанию прокрутка вверх или вниз на одну строку производится с приращением 1, которое добавляется к значению полосы прокрутки или вычитается из него. Изменить приращение можно с помощью метода `setUnitIncrement()`.

По умолчанию приращением для перелистывания страниц вверх и вниз является 10. Это значение можно изменить методом `setBlockIncrement()`. Последние два метода показаны далее.

```
void setUnitIncrement(int новыйИнкремент)
void setBlockIncrement(int новыйИнкремент)
```

Обработка полос прокрутки

Для обработки событий полосы прокрутки необходимо реализовать интерфейс `AdjustmentListener`. Как только пользователь начинает работать с полосой прокрутки, создается объект класса `AdjustmentEvent`. Его метод `getAdjustmentType()` служит для определения типа настройки. В табл. 25.1 перечислены типы событий настройки.

Таблица 25.1. События настройки

Событие	Описание
`BLOCK_DECREMENT`	Произошло событие перелистывания страницы вниз
`BLOCK_INCREMENT`	Произошло событие перелистывания страницы вверх
`TRACK`	Произошло событие абсолютного перемещения
`UNIT_DECREMENT`	Нажата кнопка перевода страницы на одну строку вниз
`UNIT_INCREMENT`	Нажата кнопка перевода страницы на одну строку вверх

Следующий код создает вертикальную и горизонтальную полосы прокрутки. Отображаются текущие настройки полос прокрутки. При перемещении курсора мыши в окне координаты каждого события перемещения будут использоваться для обновления полос прокрутки. Звездочка отображается в текущей позиции при перетаскивании. Обратите внимание на то, как метод `setPreferredSize()` позволяет устанавливать размер полос прокрутки.

```
// Демонстрация применения полос прокрутки.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SBDemo" width=300 height=200>
</applet>
*/

public class SBDemo extends Applet
implements AdjustmentListener, MouseMotionListener {
    String msg = "";
    Scrollbar vertSB, horzSB;

    public void init() {
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));
        vertSB = new Scrollbar(Scrollbar.VERTICAL,
                               0, 1, 0, height);
        vertSB.setPreferredSize(new Dimension(20, 100));

        horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
                               0, 1, 0, width);
        horzSB.setPreferredSize(new Dimension(100, 20));

        add(vertSB);
        add(horzSB);

        // регистрация на получение уведомлений о событиях
        vertSB.addAdjustmentListener(this);
        horzSB.addAdjustmentListener(this);

        addMouseMotionListener(this);
    }
```

```
    public void adjustmentValueChanged(AdjustmentEvent ae) {
    repaint();
    }

        // Обновление полос прокрутки в ответ на перетаскивание
        // с помощью мыши.
        public void mouseDragged(MouseEvent me) {
            int x = me.getX();
            int y = me.getY();
            vertSB.setValue(y);
            horzSB.setValue(x);
            repaint();
        }

    // Это нужно для MouseMotionListener
    public void mouseMoved(MouseEvent me) {
    }

    // Отображение текущего значения полос прокрутки.
    public void paint(Graphics g) {
        msg = "Vertical: " + vertSB.getValue();
     // msg = "Вертикальная: " + vertSB.getValue();
        msg += ", Horizontal: " + horzSB.getValue();
     // msg += ", Горизонтальная: " + horzSB.getValue();
        g.drawString(msg, 6, 160);

        // показываем текущую позицию при перетаскивании с помощью мыши
        g.drawString("*", horzSB.getValue(),
    vertSB.getValue());
    }
}
```

Аплет SBDemo во время выполнения показан на рис. 25.7.

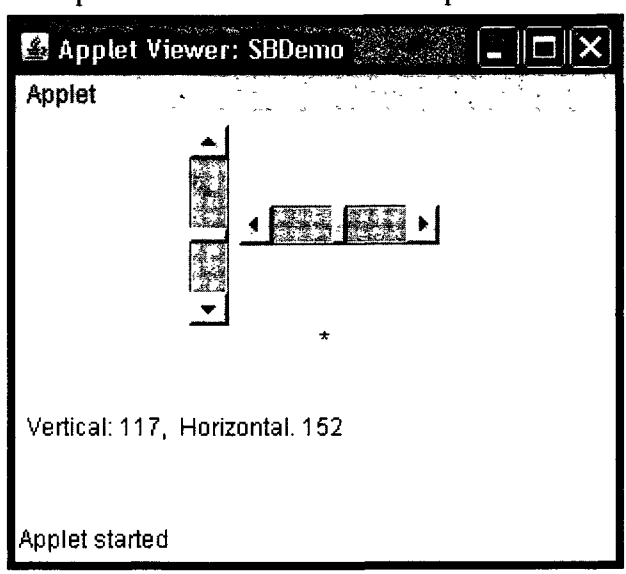


Рис. 25.7. Окно аплета SBDemo

Использование класса TextField

Класс TextField реализует однострочную область для ввода текста, которая называется *текстовым полем*. В текстовых полях пользователь может вводить строки и редактировать текст. Класс TextField является подклассом класса TextComponent и определяет следующие конструкторы.

```
TextField() throws HeadlessException
TextField(int количСимволов) throws HeadlessException
TextField(String строка) throws HeadlessException
TextField(String строка, int количСимволов) throws HeadlessException
```

Первый вариант создает текстовое поле. Второй — текстовое поле, ширина которого в символах определяется параметром *количСимволов*. Третий вариант конструктора инициализирует текстовое поле и устанавливает его ширину.

Класс TextField и его суперкласс TextComponent предлагают несколько методов, с помощью которых можно работать с текстовым полем. Чтобы узнать, какая строка содержится на данный момент в текстовом поле, вызовите метод getText(). Чтобы настроить текст, вызовите метод setText(). Эти методы показаны ниже.

```
String getText()
void setText(String строка)
```

Здесь параметр *строка* определяет новую строку.

Пользователь может выделить часть текста в текстовом поле. Часть текста также можно выделить программно с помощью метода select(). При помощи метода getSelectedText() программа сможет узнать, какой текст выделен в данный момент. Упомянутые методы показаны ниже.

```
String getSelectedText()
void select(int начИндекс, int конИндекс)
```

Метод getSelectedText() возвращает выделенный текст. Метод select() выделяет символы, начиная со значения *начИндекс* и заканчивая значением *конИндекс*-1.

С помощью метода setEditable() пользователю можно разрешить изменять содержимое текстового поля, а с помощью метода isEditable() — редактировать текст. Эти методы показаны ниже.

```
boolean isEditable()
void setEditable(boolean правкаВозможна)
```

Метод isEditable() возвращает значение true, если текст можно изменять, и значение false — если текст изменять нельзя. Если в методе setEditable() параметр *правкаВозможна* содержит значение true, то текст можно изменять, а если значение false — текст изменять нельзя.

Иногда нужно сделать так, чтобы текст, вводимый пользователем, был невидим (в частности, при вводе пароля). С помощью метода setEchoChar() можно отключить отображение вводимых символов. Этот метод задает одиночный символ, который объект класса TextField будет отображать при вводе символов (и, следовательно, реальные символы отображаться не будут). С помощью метода getEchoChar() можно узнать, включен ли этот режим для текстового поля. Получить символ отображения можно с помощью метода getEchoChar(). Эти методы показаны ниже.

```
void setEchoChar(char символ)
boolean echoCharIsSet()
char getEchoChar()
```

Здесь параметр *символ* определяет отображаемый символ. Если значением параметра *символ* будет нуль, то восстанавливается нормальное отображение на экране.

Обработка текстовых полей

Поскольку текстовые поля выполняют собственные функции редактирования, ваша программа вообще не будет реагировать на события отдельных клавиш, происходящие в текстовом поле. Однако вам, возможно, нужно будет сделать так, чтобы пользователь мог нажимать клавишу <Enter>. При нажатии этой клавиши будет происходить событие действия.

Ниже показан пример, в котором создается классический экран для ввода имени пользователя и пароля.

```
// Демонстрация применения текстового поля.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/

public class TextFieldDemo extends Applet
implements ActionListener {

    TextField name, pass;

    public void init() {
        Label namep = new Label("Name: ", Label.RIGHT);
     // Label namep = new Label("Имя: ", Label.RIGHT);
        Label passp = new Label("Password: ", Label.RIGHT);
     // Label passp = new Label("Пароль: ", Label.RIGHT);
        name = new TextField(12);
        pass = new TextField(8);
        pass.setEchoChar('?');
        add(namep);
        add(name);
        add(passp);
        add(pass);

        // регистрация на получение уведомлений о событиях действия
        name.addActionListener(this);
        pass.addActionListener(this);
    }

    // Пользователь нажал <Enter>.
    public void actionPerformed(ActionEvent ae) {
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString("Name: " + name.getText(), 6, 60);
        g.drawString("Selected text in name: " +
                     name.getSelectedText(), 6, 80);
        g.drawString("Password: " + pass.getText(), 6, 100);
     // g.drawString("Имя: " + name.getText(), 6, 60);
     // g.drawString("Выделенный текст в имени: " +
     //              name.getSelectedText(), 6, 80);
     // g.drawString("Пароль: " + pass.getText(), 6, 100);
    }
}
```

Аплет `TextFieldDemo` во время выполнения показан на рис. 25.8.

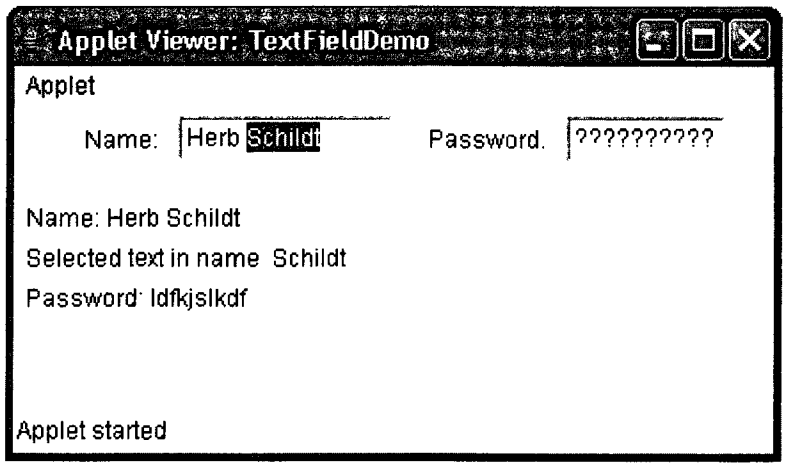


Рис. 25.8. Окно аплета `TextFieldDemo`

Использование класса `TextArea`

Иногда одной строки для ввода текста недостаточно. На этот случай библиотека AWT предлагает простой многострочный редактор — класс `TextArea`. Ниже показаны конструкторы этого класса.

```
TextArea() throws HeadlessException
TextArea(int количСтрок, int количСимволов) throws HeadlessException
TextArea(String строка) throws HeadlessException
TextArea(String строка, int количСтрок, int количСимволов) throws
HeadlessException
TextArea(String строка, int количСтрок, int количСимволов, int
пПрокрутки) throws HeadlessException
```

Здесь параметр *количСтрок* определяет высоту текстовой области, измеряемой в строках, а параметр *количСимволов* задает ее ширину, измеряемую в символах. Исходный текст можно определить при помощи параметра *строка*. В пятом варианте можно создать полосы прокрутки, которые могут понадобиться для работы с этой областью. Параметр *пПрокрутки* должен принимать одно из следующих значений.

`SCROLLBARS_BOTH`	`SCROLLBARS_NONE`
`SCROLLBARS_HORIZONTAL_ONLY`	`SCROLLBARS_VERTICAL_ONLY`

Класс `TextArea` является подклассом класса `TextComponent`. Следовательно, он поддерживает методы `getText()`, `setText()`, `getSelectedText()`, `select()`, `isEditable()` и `setEditable()`, описанные в предыдущем разделе.

Класс `TextArea` включает следующие дополнительные методы.

```
void append(String строка)
void insert(String строка, int индекс)
void replaceRange(String строка, int начИндекс, int конИндекс)
```

Метод `append()` добавляет строку, определяемую при помощи параметра *строка*, в конец текущего текста. Метод `insert()` вставляет строку, указанную параметром *строка*, в место, определенное индексом. Чтобы заменить текст, вызовите метод `replaceRange()`. Он заменяет символы, начиная с индекса, указанного в параметре *начИндекс*, и заканчивая индексом, указанным в параметре *конИндекс*`-1`, на текст, передаваемый в параметре *строка*.

Текстовые области являются практически автономными элементами управления. Вашей программе не грозят издержки, связанные с управлением областями текста. Обычно программа просто получает текущий текст, если в этом есть необходимость. Однако, при желании, вы можете прослушивать события класса `TextEvent`.

В следующей программе создается элемент управления класса `TextArea`.

```
// Демонстрация применения TextArea.
import java.awt.*;
import java.applet.*;

/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/

public class TextAreaDemo extends Applet {

    public void init() {
        String val =
            "Java 7 is the latest version of the most\n" +
            "widely-used computer language 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.";

        TextArea text = new TextArea(val, 10, 30);
        dd(text);
    }
}
```

Аплет `TextAreaDemo` во время выполнения показан на рис. 25.9.

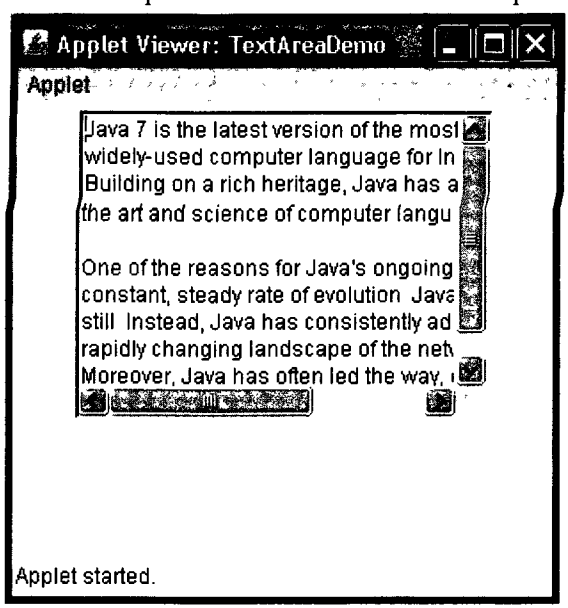


Рис. 25.9. Окно аплета `TextAreaDemo`

Диспетчеры компоновки

Каждый рассмотренный до настоящего времени компонент позиционировался диспетчером компоновки, используемым по умолчанию. Как было замечено в начале этой главы, диспетчер компоновки автоматически размещает ваши элементы управления внутри окна с применением некоторого алгоритма. Если вам приходилось писать программы для других сред с графическим пользовательским интерфейсом (например, для Windows), то вы, возможно, привыкли размещать свои средства управления вручную. Хотя элементы управления, созданные с помощью Java, тоже можно размещать вручную, вы вряд ли этим будете заниматься. На это есть две главные причины. Во-первых, размещать вручную большое количество компонентов очень утомительно. Во-вторых, в тот момент, когда нужно расположить какой-нибудь элемент управления, вы можете не знать о том, какую он имеет высоту и ширину, поскольку на этот момент еще не будут готовы собственные компоненты инструментария. Эта ситуация подобна загадке о происхождении курицы и яйца — трудно понять, когда можно использовать размеры данного компонента для его позиционирования относительно другого компонента.

У каждого объекта класса `Container` имеется свой диспетчер компоновки, который представляет собой экземпляр любого класса, реализующего интерфейс `LayoutManager`. Диспетчер компоновки устанавливается при помощи метода `setLayout()`. Если вызов метода `setLayout()` не осуществляется, используется диспетчер компоновки, принятый по умолчанию. Всякий раз при изменении размеров контейнера (или при начальном определении размеров) диспетчер компоновки используется для позиционирования каждого компонента внутри контейнера.

Метод `setLayout()` имеет следующую общую форму.

```
void setLayout(LayoutManager объектКомпоновки)
```

Здесь параметр *объектКомпоновки* представляет ссылку на требуемый диспетчер компоновки. Если вы хотите отменить использование диспетчера компоновки и позиционировать компоненты вручную, присвойте параметру *объектКомпоновки* значение `null`. Если вы сделаете это, вам нужно будет определить форму и позицию каждого компонента вручную с помощью метода `setBounds()`, определенного в классе `Component`. Обычно вы будете работать с диспетчером компоновки.

Каждый диспетчер компоновки следит за списком компонентов, хранящихся под своими именами. Диспетчер компоновки получает уведомление каждый раз, когда вы добавляете компонент в контейнер. Всякий раз, когда нужно изменить размеры контейнера, диспетчер компоновки использует для этого свои методы `minimumLayoutSize()` и `preferredLayoutSize()`. Каждый компонент, который находится под управлением диспетчера компоновки, содержит методы `getPreferredSize()` и `getMinimumSize()`. Они возвращают предпочтительный и минимальный размеры, которые необходимы для отображения каждого компонента. Диспетчер компоновки будет учитывать их, если это вообще будет возможно, и поддерживать непротиворечивую политику размещения. Можно переопределить эти методы для элементов управления, для которых вы создаете подклассы. Иначе будут применяться значения по умолчанию.

Java имеет несколько предварительно определенных классов диспетчеров компоновки, часть из которых будет описана далее. Вы можете использовать такой диспетчер компоновки, который наилучшим образом подходит для вашего приложения.

Класс `FlowLayout`

Диспетчер компоновки класса `FlowLayout` используется по умолчанию. Именно этот диспетчер компоновки применялся в предыдущих примерах. Диспетчер клас-

са FlowLayout реализует простой стиль компоновки, который подобен тому, как следуют друг за другом слова в текстовом редакторе. Направление размещения определяется свойством ориентации компонента контейнера, которое по умолчанию задает направление слева направо и сверху вниз. Поэтому по умолчанию компоненты размещаются построчно, начиная с левого верхнего угла. В любом случае, если строка больше не может уместить компонент, он появится в следующей строке. Между каждым компонентом остается небольшой промежуток: сверху и снизу, а также справа и слева. Ниже показаны конструкторы класса FlowLayout.

```
FlowLayout()
FlowLayout(int как)
FlowLayout(int как, int гориз, int верт)
```

Первый вариант размещает элементы по схеме, принятой по умолчанию, — компоненты размещаются по центру, а между ними остается промежуток, равный пяти пикселям. Второй вариант позволяет определить способ расположения каждой строки. Ниже представлены допустимые значения параметра *как*.

- FlowLayout.LEFT
- FlowLayout.CENTER
- FlowLayout.RIGHT
- FlowLayout.LEADING
- FlowLayout.TRAILING

Эти значения определяют выравнивание по левому краю, по центру, по правому краю, переднему и заднему соответственно. Третий конструктор позволяет определить промежутки по горизонтали и вертикали между компонентами при помощи параметров *гориз* и *верт*. Ниже показан вариант уже знакомого вам аплета CheckboxDemo, измененного таким образом, что в нем используется последовательное размещение с выравниванием влево.

```
// Компоновка с выравниванием влево.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="FlowLayoutDemo" width=240 height=200>
</applet>
*/

public class FlowLayoutDemo extends Applet
implements ItemListener {

    String msg = "";
    Checkbox winXP, win7, solaris, mac;

    public void init() {
        // Задаем компоновку с выравниванием влево.
        setLayout(new FlowLayout(FlowLayout.LEFT));

        winXP = new Checkbox("Windows XP", null, true);
        win7 = new Checkbox("Windows 7");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        add(winXP);
        add(win7);
        add(solaris);
```

```
        add(mac);

        // Регистрация на получение уведомлений о событиях.
        winXP.addItemListener(this);
        win7.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }

    // Перерисовываем в случае изменения состояния флажка.
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }

    // Отображаем текущее состояние флажков.
    public void paint(Graphics g) {

        msg = "Current state: ";
     // msg = "Текущее состояние: ";
        g.drawString(msg, 6, 80);
        msg = " Windows XP: " + winXP.getState();
        g.drawString(msg, 6, 100);
        msg = " Windows 7: " + win7.getState();
        g.drawString(msg, 6, 120);
        msg = " Solaris: " + solaris.getState();
        g.drawString(msg, 6, 140);
        msg = " Mac: " + mac.getState();
        g.drawString(msg, 6, 160);
    }
}
```

Аплет `FlowLayoutDemo` во время выполнения показан на рис. 25.10. Сравните его с результатом, который был получен при выполнении аплета `CheckboxDemo` (см. рис. 25.3).

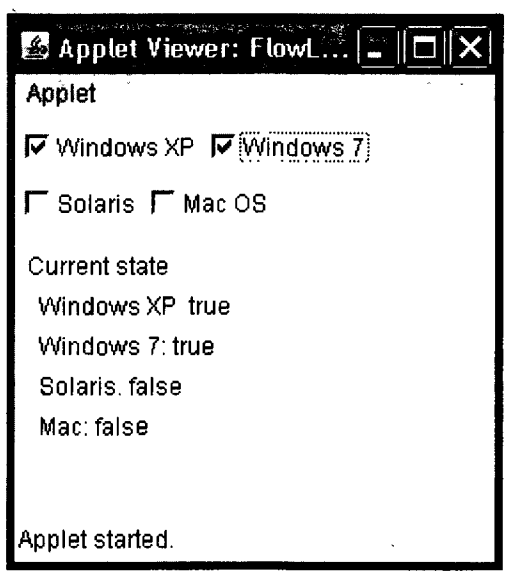


Рис. 25.10. Окно аплета `FlowLayoutDemo`

Класс BorderLayout

Этот класс реализует общий стиль компоновки для окон переднего плана. Он имеет четыре узких компонента с фиксированной шириной по краям и одну большую область в центре. Четыре стороны именуют по сторонам света: север, юг, запад и восток. Область посредине именуется центром. Ниже показаны конструкторы, определяемые классом BorderLayout.

```
BorderLayout()
BorderLayout(int гориз, int верт)
```

Первый вариант создает компоновку границы по умолчанию. Второй вариант устанавливает горизонтальный и вертикальный промежутки между компонентами при помощи параметров *гориз* и *верт*.

Класс BorderLayout определяет следующие константы, с помощью которых указываются области.

BorderLayout.CENTER	BorderLayout.SOUTH
BorderLayout.EAST	BorderLayout.WEST
BorderLayout.NORTH	

При добавлении компонентов вы будете использовать эти константы в следующей форме метода add(), который определяется в классе Container.

```
void add(Component объектУпр, Object регион)
```

Здесь параметр *объектУпр* представляет добавляемый компонент, а параметр *регион* указывает, где будет добавлен компонент.

Ниже показан пример объекта класса BorderLayout с компонентом в каждой области размещения.

```
// Демонстрация применения BorderLayout.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/

public class BorderLayoutDemo extends Applet {
    public void init() {
        setLayout(new BorderLayout());

        add(new Button("Кнопка вверху."), BorderLayout.NORTH);
        add(new Label(
              "Сюда можно поместить сообщение нижнего колонтитула."),
              BorderLayout.SOUTH);
        add(new Button("Справа"), BorderLayout.EAST);
        add(new Button("Слева"), 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);
    }
}
```

Аплет BorderLayoutDemo во время выполнения показан на рис. 25.11.

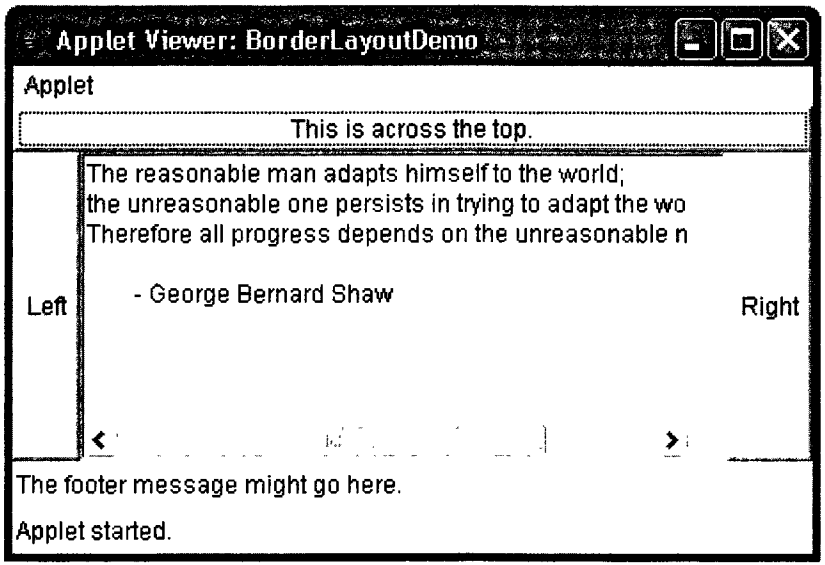


Рис. 25.11. Окно аплета BorderLayoutDemo

Использование класса Insets

Иногда нужно, чтобы между контейнером, в котором содержатся ваши компоненты, и окном, где он находится, оставался небольшой промежуток. Для этого необходимо переопределить метод getInsets(), определяемый классом Container. Эта функция возвращает объект класса Insets, который содержит верхнюю, нижнюю, левую и правую вставки (inset), используемые при отображении объекта. Эти значения диспетчер компоновки применяет для вставки компонентов во время компоновки окна. Ниже показан конструктор класса Insets.

```
Insets(int сверху, int слева, int снизу, int справа)
```

Значения, передаваемые в параметрах *сверху*, *слева*, *снизу* и *справа*, определяют промежуток между контейнером и окном, в котором он заключен. Метод getInsets() имеет следующую общую форму.

```
Insets getInsets()
```

При переопределении одного из этих методов следует вернуть новый объект класса Insets, который будет содержать требуемую промежуточную вставку.

Ниже показан измененный вариант предыдущего примера применения диспетчера класса BorderLayout. Здесь компоненты расставляются с отступом в десять пикселей. Для фона был выбран голубой цвет, чтобы можно было отличить вставки.

```
// Демонстрация применения BorderLayout и вставок.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="InsetsDemo" width=400 height=200>
</applet>
*/
```

```
public class InsetsDemo extends Applet {
    public void init() {
        // задаем цвет фона, чтобы без труда различать вставки
        setBackground(Color.cyan);

        setLayout(new BorderLayout());

        add(new Button("This is across the top."),
            BorderLayout.NORTH);
     // add(new Button("Кнопка вверху."),
     //     BorderLayout.NORTH);
        add(new Label("The footer message might go here."),
     // add(new Label(
                "Сюда можно поместить сообщение нижнего колонтитула."),
            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);
    }
    // добавляем вставки
    public Insets getInsets() {
        return new Insets(10, 10, 10, 10);
    }
}
```

Аплет `InsetsDemo` во время выполнения показан на рис. 25.12.

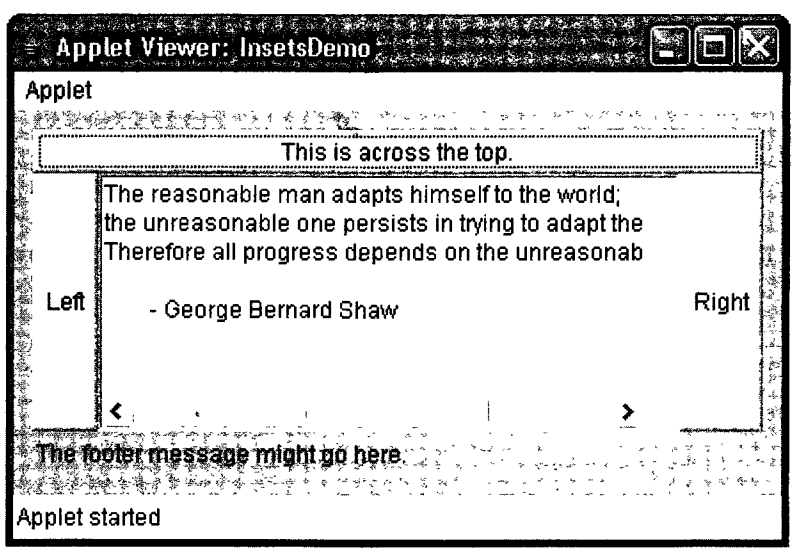


Рис. 25.12. Окно аплета `InsetsDemo`

Класс GridLayout

При использовании класса GridLayout компоненты размещаются в двухмерной сетке. Когда вы реализуете класс GridLayout, определяете количество строк и столбцов. Ниже показаны конструкторы, поддерживаемые классом GridLayout.

```
GridLayout()
GridLayout(int колСтрок, int колКолонок)
GridLayout(int колСтрок, int колКолонок, int гориз, int верт)
```

Первый вариант создает сеточную компоновку с одним столбцом. Второй вариант порождает сеточную компоновку с заданным количеством строк и столбцов. Третья форма позволяет определить горизонтальный и вертикальный промежутки между компонентами при помощи параметров *гориз* и *верт*. Один из параметров *колСтрок* и *колКолонок* может принимать нулевое значение. Если параметру *колСтрок* присвоить нулевое значение, то столбцы не будут иметь ограничения по длине. Если нулевое значение присвоить параметру *колКолонок*, строки не будут иметь ограничения по длине.

Ниже показан пример программы, которая создает сетку 4×4 и заполняет ее 15 кнопками, каждая из которых обозначается своим индексом.

```
// Демонстрация применения GridLayout
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/

public class GridLayoutDemo extends Applet {
    static final int n = 4;
    public void init() {
        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));
            }
        }
    }
}
```

Аплет GridlayoutDemo во время выполнения показан на рис. 25.13.

Совет. Этот пример можно взять за основу для написания программы игры в "пятнадцать".

Класс CardLayout

Уникальным среди остальных диспетчеров компоновки является класс CardLayout, поскольку он хранит несколько различных компоновок. Каждую компоновку можно представить в виде отдельной карточки из картотеки; эти карточки можно перетасовывать как угодно, и в любой момент в начале картотеки будет находиться какая-нибудь карточка. Такой вариант может быть полезен для пользовательских интерфейсов с необязательными компонентами, которые

можно динамически включать и отключать в зависимости от вводимых пользователем данных. Вы можете подготовить другие схемы компоновки и сделать их скрытыми, готовыми к активизации в случае необходимости.

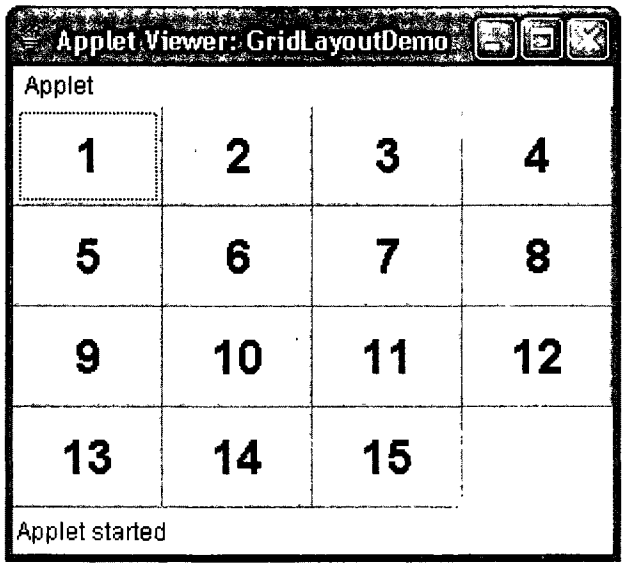


Рис. 25.13. Окно аплета `GridlayoutDemo`

Класс `CardLayout` предлагает следующие два конструктора.

```
CardLayout()
CardLayout(int гориз, int верт)
```

Первый вариант создает карточную компоновку по умолчанию. Второй вариант позволяет указать горизонтальный и вертикальный промежутки между компонентами при помощи параметров *гориз* и *верт*.

Использование карточной компоновки требует большего объема работы, чем другие схемы компоновки. Карточки обычно хранятся в объекте класса `Panel`. Для этой панели необходимо выбрать диспетчер компоновки класса `CardLayout`. Карточки, образующие картотеку, обычно тоже являются объектами класса `Panel`. Следовательно, понадобится создать панель, которая будет содержать картотеку, и панель для каждой карточки из этой картотеки. Затем нужно добавить в соответствующую панель компоненты, формирующие каждую карточку. После этого нужно добавить данную панель на главную панель аплета. Когда это будет сделано, необходимо будет подумать о том, как пользователь сможет производить свой выбор в картотеке.

При добавлении карточки на панель ей обычно присваивается имя. Поэтому в большинстве случаев будете использовать следующую форму метода `add()` для добавления карточек на панель.

```
void add(Component объектПанели, Object имя)
```

Здесь параметр *имя* представляет имя карточки, панель которой определяется при помощи параметра *объектПанели*.

После того как создадите картотеку, программа будет активизировать карточку с помощью одного из следующих методов, определяемых классом `CardLayout`.

```
void first(Container панель)
void last(Container панель)
void next(Container панель)
```

```
void previous(Container панель)
void show(Container панель, String имяКарточки)
```

Здесь параметр *панель* представляет ссылку на контейнер (обычно панель), который хранит карточки, а параметр *имяКарточки* — имя карточки. В результате вызова метода `first()` будет отображена первая карточка в картотеке. Чтобы показать последнюю карточку, вызовите метод `last()`. Для отображения следующей карточки вызовите метод `next()`. Чтобы показать предыдущую карточку, вызовите метод `previous()`. Методы `next()` и `previous()` автоматически циклически переходят по картотеке вверх или вниз соответственно. Метод `show()` отображает карточку, имя которой передается в параметре *имяКарточки*.

В следующем коде создается двухуровневая картотека, в которой пользователь может выбрать операционную систему. Операционные системы семейства Windows отображаются на одной карточке, операционные системы Mac OS и Solaris — на другой.

```
// Демонстрация применения CardLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CardLayoutDemo" width=300 height=100>
</applet>
*/

public class CardLayoutDemo extends Applet

implements ActionListener, MouseListener {
    Checkbox winXP, win7, solaris, mac;
    Panel osCards;
    CardLayout cardLO;
    Button Win, Other;

    public void init() {
        Win = new Button("Windows");
        Other = new Button("Other");
        add(Win);
        add(Other);

        cardLO = new CardLayout();
        osCards = new Panel();
        osCards.setLayout(cardLO); // присваиваем компоновке
                                   // карточки компоновку панели
        winXP = new Checkbox("Windows XP", null, true);
        win7 = new Checkbox("Windows 7");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        // добавляем на панель флажки Windows
        Panel winPan = new Panel();
        winPan.add(winXP);
        winPan.add(win7);

        // добавляем на панель флажки других ОС
        Panel otherPan = new Panel();
        otherPan.add(solaris);
        otherPan.add(mac);

        // добавляем панели в карточную панель
        osCards.add(winPan, "Windows");
```

```
        osCards.add(otherPan, "Other");

        // добавляем карточки в главную панель аплета
        add(osCards);

        // регистрация на получение уведомлений о событиях
        Win.addActionListener(this);
        Other.addActionListener(this);

        // регистрируем события мыши
        addMouseListener(this);
    }

    // Циклический перебор панелей.
    public void mousePressed(MouseEvent me) {
        cardLO.next(osCards);
    }

    // Создаем пустые заготовки для остальных методов MouseListener.
    public void mouseClicked(MouseEvent me) {
    }
    public void mouseEntered(MouseEvent me) {
    }
    public void mouseExited(MouseEvent me) {
    }
    public void mouseReleased(MouseEvent me) {
    }

    public void actionPerformed(ActionEvent ae) {
    if(ae.getSource() == Win) {
        cardLO.show(osCards, "Windows");
    }
    else {
        cardLO.show(osCards, "Other");
    }
}
```

На рис. 25.14 показан аплет `CardLayoutDemo` во время выполнения. Каждая карточка активизируется после щелчка на ее кнопке. Щелкая кнопкой мыши, можно циклически переходить от одной карточки к другой.

Класс `GridBagLayout`

Хотя предыдущие схемы компоновки отлично подходят для использования во многих аплетах, для некоторых из них все же необходим подробный контроль расположения компонентов в окне. Для этого удобно применять сеточную компоновку (grid bag), которая определена в классе `GridBagLayout`. Компоновка "grid bag" обладает полезным свойством, которое позволяет задавать относительное расположение компонентов, указывая их позиции в ячейках сетки. Ключевой особенностью компоновки "grid bag" является то, что каждый компонент может иметь свои размеры, а каждая строка в сетке — свое количество столбцов. Этим и объясняется название этой схемы компоновки — *grid bag* ("сетка-мешок"). Это — коллекция небольших соединенных вместе сеток.

Местонахождение и размеры каждого компонента в сеточной компоновке определяются набором связанных с ним ограничений, которые содержатся в объекте класса `GridBagConstraints`. Ограничения включают высоту и ширину ячейки, расположение компонента, его выравнивание и точку связывания внутри ячейки.

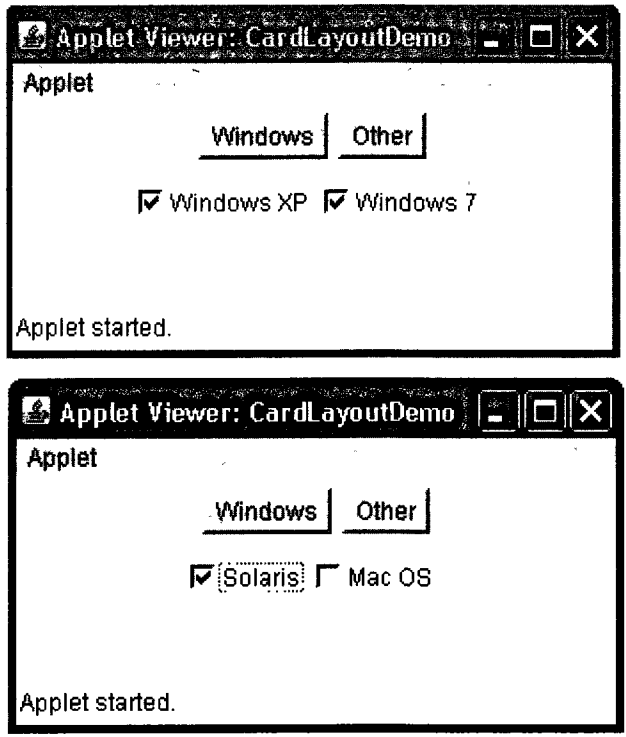


Рис. 25.14. Аплет `CardLayoutDemo` во время выполнения

Общая процедура использования сеточной компоновки выглядит так. Сначала создается новый объект класса `GridBagLayout`, который определяется как текущий диспетчер компоновки. Затем выбираются ограничения для каждого компонента, добавляемого в сетку. После этого компоненты добавляются к диспетчеру компоновки. Хотя класс `GridBagLayout` является более сложным по сравнению с другими диспетчерами компоновки, использовать его будет очень легко, если вы хорошенько разберетесь с принципом его работы.

Класс `GridBagLayout` определяет только один конструктор.

```
GridBagLayout()
```

Кроме того, этот класс определяет несколько методов, многие из которых являются защищенными и не предназначены для общего использования. Однако есть один метод, который следует использовать, — `setConstraints()`.

```
void setConstraints(Component компаратор, GridBagConstraints
ограничения)
```

Здесь параметр *компаратор* представляет компонент, к которому применяются ограничения, определяемые с помощью параметра *ограничения*. Этот метод описывает ограничения, применяемые к каждому компоненту в сетке.

Залогом успешного использования класса `GridBagLayout` является тщательная настройка его ограничений, которые хранятся в объекте класса `GridBagConstraints`. Класс `GridBagConstraints` определяет несколько полей, которые вы можете настроить для управления размерами, размещением и промежутками компонента. Эти поля перечислены в табл. 25.2. Некоторые из них будут описаны подробно немного позже.

Таблица 25.2. Ограничительные поля, определяемые в классе GridBagConstraints

Поле	Назначение
int anchor	Задает местоположение компонента внутри ячейки. Значение по умолчанию — GridBagConstraints.CENTER
int fill	Задает способ изменения размеров компонента, если размеры компонента меньше размеров его ячейки. Допустимыми являются значения GridBagConstraints.NONE (по умолчанию), GridBagConstraints.HORIZONTAL, GridBagConstraints.VERTICAL и GridBagConstraints.BOTH
int gridheight	Задает высоту компонента в пересчете на ячейки. Значение по умолчанию — 1
int gridwidth	Задает ширину компонента в пересчете на ячейки. Значение по умолчанию — 1
int gridx	Задает координату X ячейки, в которую будет добавлен компонент. Значение по умолчанию — GridBagConstraints.RELATIVE
int gridy	Задает координату Y ячейки, в которую будет добавлен компонент. Значение по умолчанию — GridBagConstraints.RELATIVE
Insets insets	Задает вставки. По умолчанию все вставки являются нулевыми
int ipadx	Задает дополнительный горизонтальный промежуток, окружающий компонент внутри ячейки. Значение по умолчанию — 0
int ipady	Задает дополнительный вертикальный промежуток, окружающий компонент внутри ячейки. Значение по умолчанию — 0
double weightx	Задает весовое значение, которое определяет горизонтальные промежутки между ячейками и краями контейнера, в котором они содержатся. Значение по умолчанию — 0.0. Чем больше вес, тем больше будет промежуток. Если все значения будут равны 0.0, то дополнительные промежутки будут распределены равномерно между краями окна
double weighty	Задает весовое значение, которое определяет вертикальные промежутки между ячейками и краями контейнера, в котором они содержатся. Значение по умолчанию — 0.0. Чем больше вес, тем больше будет промежуток. Если все значения будут равны 0.0, то дополнительные промежутки будут распределены равномерно между краями окна

Класс GridBagConstraints определяет также несколько статических полей, которые содержат стандартные значения ограничений, такие как значения GridBagConstraints.CENTER и GridBagConstraints.VERTICAL.

Если размеры компонента меньше размеров его ячейки, можно воспользоваться полем anchor, чтобы определить, где внутри ячейки будет располагаться верхний левый угол компонента. Существует три типа значений, которые можно присвоить полю anchor. Первые являются абсолютными значениями.

GridBagConstraints.CENTER	GridBagConstraints.SOUTH
GridBagConstraints.EAST	GridBagConstraints.SOUTHEAST
GridBagConstraints.NORTH	GridBagConstraints.SOUTHWEST
GridBagConstraints.NORTHEAST	GridBagConstraints.WEST
GridBagConstraints.NORTHWEST	

Как можно судить по именам этих значений, они определяют расположение компонента в определенных местах.

Второй тип значений, которые можно присвоить полю anchor, является относительным, т.е. относительно ориентации контейнера, которая в восточных языках может быть другой. 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.

Третий тип значений, которые могут быть присвоены полю anchor, позволяет позиционировать компоненты вертикально по отношению к базовой линии строки. Эти значения перечислены ниже.

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	

При горизонтальном положении центрирование может осуществляться по отношению или к переднему краю (LEADING), или к заднему (TRAILING).

Поля weightx и weighty являются достаточно важными и на первый взгляд кажутся довольно запутанными. Их значения определяют, сколько дополнительного пространства будет выделено внутри контейнера для каждой строки и каждого столбца. По умолчанию оба поля имеют нулевые значения. Если все значения в столбце и строке будут нулевыми, то дополнительный промежуток будет распределен равномерно между краями окна. Увеличивая вес, вы увеличите это распределение пространства строки или столбца пропорционально остальным строкам или столбцам. Самый лучший способ уяснить эти значения — поэкспериментировать с ними на конкретном примере.

С помощью переменной gridwidth можно задать ширину ячейки в пересчете на единицы ячейки (cell unit). По умолчанию эта переменная имеет значение 1.

Чтобы компонент использовал свободное пространство в строке, применяйте значение GridBagConstraints.REMAINDER. Чтобы компонент мог использовать предпоследнюю ячейку в строке, применяйте значение GridBagConstraints.RELATIVE. Ограничение gridheight работает точно так же, но в вертикальном направлении.

Можно определить значение заполнения, которое будет использоваться для увеличения минимального размера ячейки. Для горизонтального заполнения присвойте значение переменной ipadx. Для вертикального заполнения присвойте значение переменной idpay.

Ниже показан пример, в котором класс GridBagLayout служит для демонстрации только что рассмотренного материала.

```
// Демонстрация применения GridBagLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
```

```
/*
<applet code="GridBagDemo" width=250 height=200>
</applet>
*/

public class GridBagDemo extends Applet
implements ItemListener {

    String msg = "";
    Checkbox winXP, win7, solaris, mac;

    public void init() {
        GridBagLayout gbag = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(gbag);

        // Определяем флажки.
        winXP = new Checkbox("Windows XP", null, true);
        win7 = new Checkbox("Windows 7");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        // Определяем сетку.

        // Первой строке задаем нулевой вес, используемый по умолчанию.
        gbc.weightx = 1.0; // используем вес столбца 1
        gbc.ipadx = 200;   // заполняем на 200 единиц
        gbc.insets = new Insets(4, 4, 0, 0); // вставка чуть в стороне
                                             // от левого верхнего угла
        gbc.anchor = GridBagConstraints.NORTHEAST;
        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(winXP, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(win7, gbc);

        // Задаем второй строке вес 1.
        gbc.weighty = 1.0;
        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(solaris, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(mac, gbc);

        // Добавляем компоненты.
        add(winXP);
        add(win7);
        add(solaris);
        add(mac);

        // Регистрация на получение уведомлений о событиях элемента.
        winXP.addItemListener(this);
        win7.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }
    // Перерисовка в случае изменения состояния флажка.
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
    // Отображаем текущее состояние флажков.
    public void paint(Graphics g) {
        msg = "Current state: ";
     // msg = "Текущее состояние: ";
        g.drawString(msg, 6, 80);
    msg = " Windows XP: " + winXP.getState();
```

```
    g.drawString(msg, 6, 100);
    msg = " Windows 7: " + win7.getState();
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = " Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
    }
}
```

Аплет `GridBagDemo` во время выполнения показан на рис. 25.15.

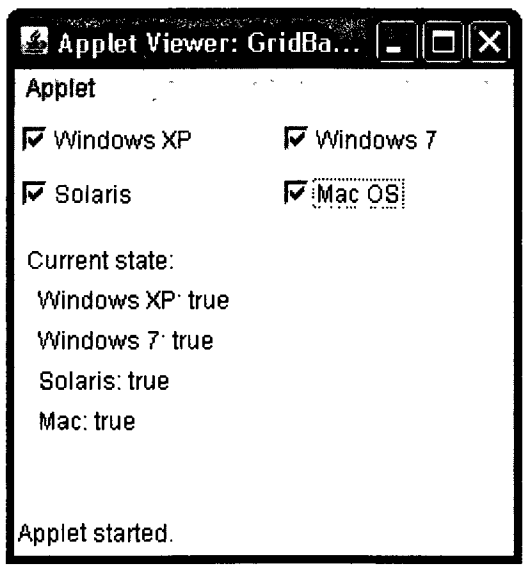


Рис. 25.15. Окно аплета `GridBagDemo`

В этой схеме компоновки флажки операционных систем размещаются в сетке 2×2. Каждая ячейка имеет заполнение 200. Каждый компонент вставляется рядом с верхним левым углом (в 4 единицах от него). Вес столбца составляет 1, благодаря чему любой дополнительный горизонтальный промежуток распределяется равномерно между столбцами. Первая строка использует вес по умолчанию, равный 0; вторая строка имеет вес 1. Это означает, что любой дополнительный вертикальный промежуток переносится во вторую строку.

Класс `GridBagLayout` является очень мощным диспетчером компоновки. Стоит потратить некоторое время на его изучение и эксперименты с ним. После того как поймете, для каких целей предназначены различные настройки, сможете использовать класс `GridBagLayout` для расположения компонентов с высокой степенью точности.

Полосы меню и меню

Окно переднего плана может иметь связанную с ним полосу меню, которая отображает список пунктов меню верхнего уровня. Каждый пункт меню связан с выпадающим меню. Этот принцип реализован в библиотеке AWT с помощью классов `MenuBar`, `Menu` и `MenuItem`. В общем случае полоса меню состоит из одно-

го или нескольких объектов класса Menu. Каждый объект класса Menu содержит список объектов класса MenuItem. Каждый объект класса MenuItem представляет то, что может выбрать пользователь. Поскольку класс Menu является подклассом класса MenuItem, можно создать иерархию вложенных подменю. Можно также включать отмечаемые пункты меню, которые являются пунктами меню типа CheckboxMenuItem; если пользователь щелкнет на таком пункте, рядом с ним появится отметка (в виде "галочки").

Чтобы создать полосу меню, сначала создайте экземпляр класса MenuBar. Этот класс определяет только конструктор, который используется по умолчанию. Затем создайте экземпляры класса Menu, которые будут определять варианты выбора, отображаемые в строке. Ниже показаны конструкторы класса Menu.

```
Menu() throws HeadlessException
Menu(String имяПункта) throws HeadlessException
Menu(String имяПункта, boolean удаляемый) throws HeadlessException
```

Здесь *имяПункта* представляет имя пункта меню. Если параметр *удаляемый* содержит значение true, меню можно удалять и очищать. В противном случае оно останется присоединенным к полосе меню. (Удаляемые меню зависят от реализации.) Первый вариант создает пустое меню.

Отдельные пункты меню имеют тип MenuItem. Он определяет следующие конструкторы.

```
MenuItem() throws HeadlessException
MenuItem(String имяПункта) throws HeadlessException
MenuItem(String имяПункта, MenuShortcut клавишаДоступа) throws
HeadlessException
```

Здесь параметр *имяПункта* представляет имя, отображаемое в меню, а параметр *клавишаДоступа* — клавишу быстрого доступа для данного пункта меню.

Пункт меню можно включить или отключить с помощью метода setEnabled().

```
void setEnabled(boolean флагРазрешения)
```

Если параметр *флагРазрешения* содержит значение true, пункт меню будет включен, а если false — отключен.

Состояние пункта меню можно определить с помощью метода isEnabled().

```
boolean isEnabled()
```

Метод isEnabled() возвращает значение true, если пункт меню, для которого он вызывается, является включенным. В противном случае он возвращает значение false.

Изменить имя пункта меню можно с помощью метода setLabel(). Текущее имя пункта меню можно узнать с помощью метода getLabel(). Эти методы показаны далее.

```
void setLabel(String новоеИмя)
String getLabel()
```

Здесь параметр *новоеИмя* представляет новое имя вызываемого пункта меню. Метод getLabel() возвращает текущее имя.

Создать отмечаемый пункт меню можно с помощью подкласса CheckboxMenuItem класса MenuItem. Он имеет следующие конструкторы.

```
CheckboxMenuItem() throws HeadlessException
CheckboxMenuItem(String имяПункта) throws HeadlessException
CheckboxMenuItem(String имяПункта, boolean вкл) throws HeadlessException
```

Здесь *имяПункта* представляет имя, отображаемое в меню. Отмечаемые пункты меню подобны триггеру (со счетным входом). Каждый раз, когда напротив одного из пунктов ставится отметка, его состояние изменяется. В первых двух формах отмечаемое поле не имеет метки. В третьей форме, если параметр *вкл* содержит значение true, отмечаемое поле имеет метку. В противном случае оно остается пустым.

Состояние отмечаемого пункта меню можно узнать с помощью метода getState(). Присвоить ему определенное состояние можно при помощи метода setState(). Эти методы показаны ниже.

```
boolean getState()
void setState(boolean вкл)
```

Если пункт меню отмечен, метод getState() возвращает значение true. В противном случае он возвращает значение false. Чтобы отметить пункт меню, передайте значение true методу setState(). Для очистки пункта меню передайте значение false.

После того как создадите пункт меню, необходимо добавить его в объект класса Menu с помощью метода add(), который имеет следующую общую форму.

```
MenuItem add(MenuItem пункт)
```

Здесь *пункт* представляет добавляемый пункт меню. Добавление пунктов в меню производится в том порядке, в каком осуществлялись вызовы метода add(). Возвращаемым значением является *пункт*.

После того как добавите все пункты в объект класса Menu, его можно будет добавить в полосу меню с помощью следующей версии метода add(), определенной в классе MenuBar.

```
Menu add(Menu меню)
```

Здесь параметр *меню* представляет добавляемое меню. Возвращаемым значением является *меню*.

События в меню происходят только при выборе пункта типа MenuItem и CheckboxMenuItem. Они не происходят, например, при обращении к полосе меню для отображения выпадающего меню. Каждый раз при выборе пункта меню создается объект класса ActionEvent. По умолчанию строка команды действия представляет собой имя элемента меню. Однако вы можете определить другую строку команды действия, вызвав метод setActionCommand() в элементе меню. Каждый раз, когда отмечается пункт меню или снимается отметка с него, создается объект класса ItemEvent. Таким образом, для обработки этих событий меню необходимо реализовать интерфейсы ActionListener и ItemListener.

Метод getItem() класса ItemEvent возвращает ссылку на пункт меню, известивший о данном событии. Ниже показана общая форма этого метода.

```
Object getItem()
```

Ниже представлен пример, который добавляет серию вложенных меню во всплывающее меню. В окне отображается выбранный пункт меню. Кроме того, отображается также состояние двух пунктов меню флажков.

```
// Пример работы с меню.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MenuDemo" width=250 height=250>
</applet>
*/

// Создаем подкласс Frame.
class MenuFrame extends Frame {
    String msg = "";
    CheckboxMenuItem debug, test;

    MenuFrame(String title) {
        super(title);

        // Создание полосы меню и добавление ее в фрейм
```

```
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);

        // создание элементов 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);

        // отмечаемые элементы меню
        debug = new CheckboxMenuItem("Debug");
        edit.add(debug);
        test = new CheckboxMenuItem("Testing");
        edit.add(test);

        mbar.add(edit);

        // создание объекта для обработки событий действия и элемента
        MyMenuHandler handler = new MyMenuHandler(this);
        // регистрируем объект на получение уведомлений об этих событиях
        item1.addActionListener(handler);
        item2.addActionListener(handler);
        item3.addActionListener(handler);
        item4.addActionListener(handler);
        item5.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);

        // создание объекта для обработки событий окна
        MyWindowAdapter adapter = new MyWindowAdapter(this);

        // регистрируем объект на получение уведомлений об этих событиях
        addWindowListener(adapter);
    }

    public void paint(Graphics g) {
        g.drawString(msg, 10, 200);

        if(debug.getState())
```

```
            g.drawString("Debug is on.", 10, 220);
        else
            g.drawString("Debug is off.", 10, 220);

        if(test.getState())
            g.drawString("Testing is on.", 10, 240);
        else
            g.drawString("Testing is off.", 10, 240);
    }
}
class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;
    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }

    public void windowClosing(WindowEvent we) {
        menuFrame.setVisible(false);
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;
    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    // Обработка событий действий.
    public void actionPerformed(ActionEvent ae) {
        String msg = "You selected ";
        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("Quit..."))
            msg += "Quit.";
        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.";
        menuFrame.msg = msg;
        menuFrame.repaint();
    }
    // Обработка событий элемента.
    public void itemStateChanged(ItemEvent ie) {
        menuFrame.repaint();
    }
}
```

```
// Создание обрамляющего окна.
public class MenuDemo extends Applet {
    Frame f;
    public void init() {
        f = new MenuFrame("Menu Demo");
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));

        setSize(new Dimension(width, height));

        f.setSize(width, height);
        f.setVisible(true);
    }

    public void start() {
        f.setVisible(true);
    }

    public void stop() {
        f.setVisible(false);
    }
}
```

Аплет MenuDemo во время выполнения показан на рис. 25.16.

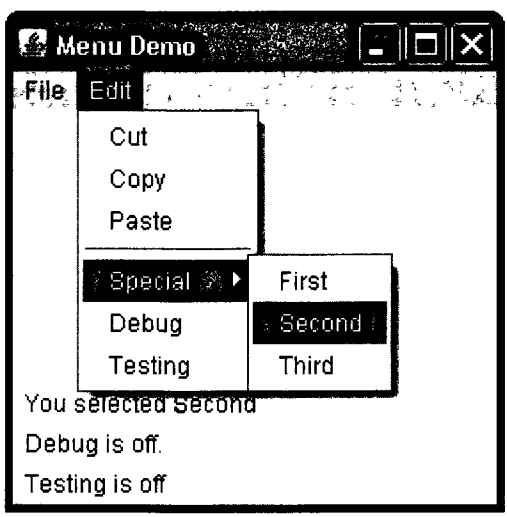


Рис. 25.16. Окно аплета MenuDemo

Существует еще один класс, обслуживающий меню, который вас может заинтересовать, — класс PopupMenu. Он работает подобно классу Menu, но формирует меню, которое отображается в определенном месте. Класс PopupMenu (контекстное меню) предлагает гибкую и полезную альтернативу некоторым типам организации меню.

Диалоговые окна

Довольно часто возникает необходимость в *диалоговом окне* для хранения набора связанных между собой элементов управления. Диалоговые окна используются, в первую очередь, для получения данных, вводимых пользователем. Нередко они

являются дочерними окнами в окне верхнего уровня. Диалоговые окна не имеют полос меню, однако в остальном они функционируют подобно обрамляющим окнам. (Например, вы можете добавлять в них элементы управления точно так же, как добавляете их в обрамляющее окно.) Диалоговые окна могут быть модальными или немодальными. Когда активным является *модальное* диалоговое окно, то все вводимые данные направляются в него до тех пор, пока оно остается открытым. Это значит, что вы не сможете обратиться к остальным частям своей программы до тех пор, пока не будет закрыто диалоговое окно. Если активным является *немодальное* диалоговое окно, то фокус ввода может быть передан другому окну вашей программы. Таким образом, остальные части вашей программы остаются активными и доступными. Диалоговые окна имеют тип `Dialog`. Ниже показаны два наиболее часто используемых конструктора.

```
Dialog(Frame родительскоеОкно, boolean режим)
Dialog(Frame родительскоеОкно, String заголовок, boolean режим)
```

Здесь параметр *родительскоеОкно* представляет "хозяина" диалогового окна. Если параметр *режим* содержит значение `true`, диалоговое окно является модальным. В противном случае оно немодальное. Заголовок диалогового окна можно указать в параметре *заголовок*. В общем случае вы будете организовывать подклассы класса `Dialog`, добавляя функциональные особенности, необходимые для вашего приложения.

Ниже показан видоизмененный вариант предыдущей программы для работы с меню, в котором отображается немодальное диалоговое окно при выборе пункта меню **New** (Новое). Обратите внимание на то, что в момент закрытия диалогового окна вызывается метод `dispose()`. Этот метод определяется классом `Window` и освобождает все системные ресурсы, задействованные диалоговым окном.

```
// Пример диалогового окна.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="DialogDemo" width=250 height=250>
</applet>
*/

// Создание подкласса Dialog.
class SampleDialog extends Dialog implements ActionListener {
    SampleDialog(Frame parent, String title) {
        super(parent, title, false);
       setLayout(new FlowLayout());
       setSize(300, 200);

       add(new Label("Press this button:"));
       Button b;
       add(b = new Button("Cancel"));
       b.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae) {
       dispose();
    }

    public void paint(Graphics g) {
       g.drawString("This is in the dialog box", 10, 70);
    }
}

// Создание подкласса Frame.
class MenuFrame extends Frame {
```

```
    String msg = "";
    CheckboxMenuItem debug, test;

    MenuFrame(String title) {
        super(title);

        // создание полосы меню и добавление ее во фрейм
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);

        // создание элементов меню
        Menu file = new Menu("File");
        MenuItem item1, item2, item3, item4;
        file.add(item1 = new MenuItem("New..."));
        file.add(item2 = new MenuItem("Open..."));
        file.add(item3 = new MenuItem("Close"));
        file.add(new MenuItem("-"));
        file.add(item4 = new MenuItem("Quit..."));
        mbar.add(file);

        Menu edit = new Menu("Edit");
        MenuItem item5, item6, item7;
        edit.add(item5 = new MenuItem("Cut"));
        edit.add(item6 = new MenuItem("Copy"));
        edit.add(item7 = new MenuItem("Paste"));
        edit.add(new MenuItem("-"));

        Menu sub = new Menu("Special", true);
        MenuItem item8, item9, item10;
        sub.add(item8 = new MenuItem("First"));
        sub.add(item9 = new MenuItem("Second"));
        sub.add(item10 = new MenuItem("Third"));
        edit.add(sub);

        // отмечаемые элементы меню
        debug = new CheckboxMenuItem("Debug");
        edit.add(debug);
        test = new CheckboxMenuItem("Testing");
        edit.add(test);

        mbar.add(edit);

        // создание объекта для обработки событий действия и элемента
        MyMenuHandler handler = new MyMenuHandler(this);
        // регистрация на получение уведомлений об этих событиях
        item1.addActionListener(handler);
        item2.addActionListener(handler);
        item3.addActionListener(handler);
        item4.addActionListener(handler);
        item5.addActionListener(handler);
        item6.addActionListener(handler);
        item7.addActionListener(handler);
        item8.addActionListener(handler);
        item9.addActionListener(handler);
        item10.addActionListener(handler);
        debug.addItemListener(handler);
        test.addItemListener(handler);

        // создание объекта для обработки событий окна
        MyWindowAdapter adapter = new MyWindowAdapter(this);
        // регистрация на получение уведомлений об этих событиях
        addWindowListener(adapter);
    }
    public void paint(Graphics g) {
```

```
        g.drawString(msg, 10, 200);
        if(debug.getState())
            g.drawString("Debug is on.", 10, 220);
        else
            g.drawString("Debug is off.", 10, 220);

        if(test.getState())
            g.drawString("Testing is on.", 10, 240);
        else
            g.drawString("Testing is off.", 10, 240);
    }
}

class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;
    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    public void windowClosing(WindowEvent we) {
        menuFrame.dispose();
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;
    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    // Обработка событий действий.
    public void actionPerformed(ActionEvent ae) {
        String msg = "You selected ";
        String arg = ae.getActionCommand();
        // Активизация диалогового окна при выборе пункта New.
        if(arg.equals("New...")) {
            msg += "New.";
            SampleDialog d = new
            SampleDialog(menuFrame, "New Dialog Box");
            d.setVisible(true);
        }
        // Определяем остальные диалоговые окна для следующих пунктов.
        else if(arg.equals("Open..."))
            msg += "Open.";
        else if(arg.equals("Close"))
            msg += "Close.";
        else if(arg.equals("Quit..."))
            msg += "Quit.";
        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.";
```

```
        menuFrame.msg = msg;
        menuFrame.repaint();
    }

    public void itemStateChanged(ItemEvent ie) {
        menuFrame.repaint();
    }
}

// Создание обрамляющего окна.
public class DialogDemo extends Applet {
    Frame f;

    public void init() {
        f = new MenuFrame("Menu Demo");
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));

        setSize(width, height);

        f.setSize(width, height);
        f.setVisible(true);
    }

    public void start() {
        f.setVisible(true);
    }

    public void stop() {
        f.setVisible(false);
    }
}
```

Аплет `DialogDemo` во время выполнения показан на рис. 25.17.

Совет. При желании можно попробовать определить диалоговые окна для других пунктов, представляемых с помощью меню.

Класс `FileDialog`

Язык Java предлагает встроенное диалоговое окно, в котором пользователь может выбрать файл. Чтобы создать диалоговое окно выбора файла, реализуйте объект класса `FileDialog`. После этого будет отображено диалоговое окно выбора файла. Обычно оно представляет собой стандартное диалоговое окно выбора файла, используемое в операционной системе. Ниже показаны конструкторы класса `FileDialog`.

```
FileDialog(Frame родительский, String заголовокОкна)
FileDialog(Frame родительский, String заголовокОкна, int как)
FileDialog(Frame родительский)
```

Здесь параметр *родительский* представляет владельца диалогового окна, а параметр *заголовокОкна* задает имя, отображаемое в строке заголовка диалогового окна. Если параметр *заголовокОкна* опустить, заголовок диалогового окна отображаться не будет. Если параметр *как* будет содержать значение `FileDialog.LOAD`, при выборе файла окно будет использоваться для чтения, а если значение `FileDialog.SAVE` — для записи. Если этот параметр опустить, окно будет использоваться при выборе файла для чтения.

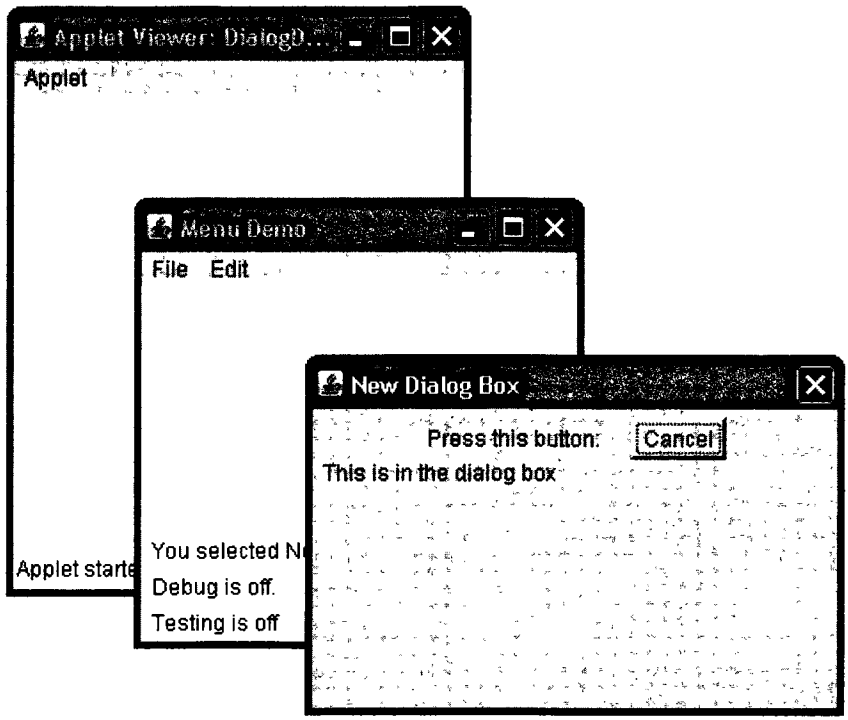


Рис. 25.17. Аплет `DialogDemo` во время выполнения

Класс `FileDialog` предлагает методы, позволяющие определять имя файла и его путь, после того как этот файл будет выбран пользователем. Ниже показано два примера этих методов.

```
String getDirectory()
String getFile()
```

Эти методы возвращают, соответственно, каталог и имя файла.

В следующей программе активизируется стандартное диалоговое окно выбора файла.

```
/* Пример диалогового окна выбора файла.
   Это — приложение, а не аплет.
*/
import java.awt.*;
import java.awt.event.*;

// Создание подкласса Frame.
class SampleFrame extends Frame {
    SampleFrame(String title) {
        super(title);

        // удаление окна после его закрытия
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }
```

```
// Пример FileDialog.
class FileDialogDemo {
    public static void main(String args[]) {
        // создание фрейма, которому будет принадлежать диалоговое окно
        Frame f = new SampleFrame("File Dialog Demo");
        f.setVisible(true);
        f.setSize(100, 100);

        FileDialog fd = new FileDialog(f, "File Dialog");

        fd.setVisible(true);
    }
}
```

На рис. 25.18 можно видеть результат выполнения этой программы. (Точная конфигурация диалогового окна может быть иной.)

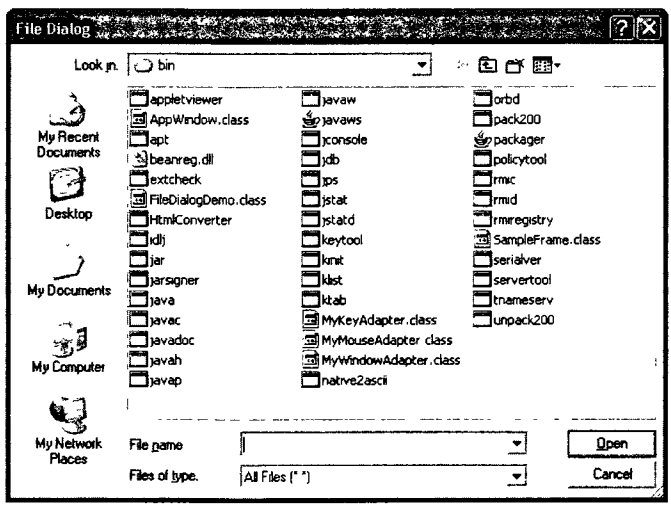


Рис. 25.18. Программа, в которой используется диалоговое окно выбора файла, во время выполнения

Последний момент: начиная с JDK 7 вы можете использовать класс `FileDialog` для выбора в списке файлов. Эти функциональные возможности обеспечивают методы `setMultipleMode()`, `isMultipleMode()` и `getFiles()`.

Обработка событий при расширении компонентов библиотеки AWT

Модель делегирования событий была описана в главе 23, и во всех предыдущих программах из этой книги использовалась именно она. Однако в Java события можно также обрабатывать за счет организации подклассов компонентов библиотеки AWT. При таком подходе можно обрабатывать события почти так же, как если бы они обрабатывались под управлением исходной версии Java 1.0. Естественно, что такой подход не рекомендуется, поскольку он имеет те же недостатки, что и модель событий в Java 1.0, среди которых главным является неэффективность. Обработка событий за счет расширения компонентов библиотеки AWT описыва-

ется в настоящем разделе для полноты рассмотрения материала. Однако в остальных разделах этой книги данный подход не используется. При расширении компонента AWT необходимо вызывать метод `enableEvents()` класса `Component`. Он имеет следующую общую форму.

```
protected final void enableEvents(long маскаСобытия)
```

Параметр *маскаСобытия* представляет битовую маску, определяющую событие, которое необходимо направить этому компоненту. Для формирования маски класс `AWTEvent` определяет константы `int`. Ниже перечислены некоторые из них.

ACTION_EVENT_MASK	KEY_EVENT_MASK
ADJUSTMENT_EVENT_MASK	MOUSE_EVENT_MASK
COMPONENT_EVENT_MASK	MOUSE_MOTION_EVENT_MASK
CONTAINER_EVENT_MASK	MOUSE_MOTION_EVENT_MASK
FOCUS_EVENT_MASK	TEXT_EVENT_MASK
INPUT_METHOD_EVENT_MASK	WINDOW_EVENT_MASK
ITEM_EVENT_MASK	

Чтобы обработать событие, следует заменить соответствующий метод одного из ваших суперклассов. Удостоверьтесь также, что вызвали версию суперкласса метода. В табл. 25.3 перечислены некоторые наиболее часто используемые методы и классы, которые их предлагают.

Таблица 25.3. Методы обработки событий

Класс	Методы обработки
`Button`	`processActionEvent()`
`Checkbox`	`processItemEvent()`
`CheckboxMenuItem`	`processItemEvent()`
`Choice`	`processItemEvent()`
`Component`	`processComponentEvent()`, `processFocusEvent()`, `processKeyEvent()`, `processMouseEvent()`, `processMouseMotionEvent()`, `processMouseWheelEvent()`
`List`	`processActionEvent()`, `processItemEvent()`
`MenuItem`	`processActionEvent()`
`Scrollbar`	`processAdjustmentEvent()`
`TextComponent`	`processTextEvent()`

В следующих разделах будут представлены простые программы, демонстрирующие расширение компонентов AWT.

Расширение класса `Button`

В приведенной ниже программе создается аплет, который отображает кнопку с меткой `"Test Button"` ("Тестовая кнопка"). Если щелкнуть на этой кнопке, в строке состояния средства просмотра аплетов или браузера будет выведена строка `"action event"` ("событие действия"), за которой последует номер нажатой кнопки.

Программа содержит один класс верхнего уровня `ButtonDemo2`, который расширяет класс `Applet`. В коде определяется статическая целочисленная переменная `i`, которой присваивается нулевое значение. Она регистрирует количество щелчков на кнопке. Метод `init()` реализует класс `MyButton` и добавляет его в аплет.

Класс MyButton является внутренним классом, расширяющим класс Button. Его конструктор использует параметр super для передачи метки кнопки конструктору суперкласса. Он вызывает метод enableEvents(), поэтому события действия могут быть получены данным объектом. Когда происходит событие действия, вызывается метод processActionEvent(). Этот метод отображает текст в строке состояния и вызывает метод progressActionEvent() для суперкласса. Поскольку класс MyButton является внутренним, он может напрямую обращаться к методу showStatus() класса ButtonDemo2.

```
/*
* <applet code=ButtonDemo2 width=200 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ButtonDemo2 extends Applet {
    MyButton myButton;
    static int i = 0;
    public void init() {
        myButton = new MyButton("Test Button");
        add(myButton);
    }
    class MyButton extends Button {
        public MyButton(String label) {
        super(label);
            enableEvents(AWTEvent.ACTION_EVENT_MASK);
        }
        protected void processActionEvent(ActionEvent ae) {
           showStatus("action event: " + i++);
            super.processActionEvent(ae);
        }
    }
}
```

Расширение класса Checkbox

В следующей программе создается аплет, который отображает три флажка с метками "Item 1", "Item 2" и "Item 3". При установке или сбросе флажка строка, содержащая имя и состояние этого флажка, будет отображаться в строке состояния средства просмотра аплетов или браузера.

Программа содержит один класс верхнего уровня CheckboxDemo2, который расширяет класс Applet. Его метод init() создает три экземпляра класса MyCheckbox и добавляет их в аплет. Класс MyCheckbox является внутренним классом, расширяющим класс Checkbox. Его конструктор использует параметр super для передачи метки флажка конструктору суперкласса. Он вызывает метод enableEvents(), так что события пунктов меню могут быть получены этим объектом. Если происходит событие пункта, вызывается метод processItemEvent(). Этот метод отображает текст в строке состояния и вызывает метод processItemEvent() для суперкласса.

```
/*
* <applet code=CheckboxDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;

public class CheckboxDemo2 extends Applet {
    MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
    public void init() {
        myCheckbox1 = new MyCheckbox("Item 1");
     // myCheckbox1 = new MyCheckbox("Элемент 1");
        add(myCheckbox1);
        myCheckbox2 = new MyCheckbox("Item 2");
     // myCheckbox2 = new MyCheckbox("Элемент 2");
        add(myCheckbox2);
        myCheckbox3 = new MyCheckbox("Item 3");
     // myCheckbox3 = new MyCheckbox("Элемент 3");
        add(myCheckbox3);
    }

    class MyCheckbox extends Checkbox {
        public MyCheckbox(String label) {
            super(label);
            enableEvents(AWTEvent.ITEM_EVENT_MASK);
        }
        protected void processItemEvent(ItemEvent ie) {
            showStatus("Checkbox name/state: " + getLabel() +
                       "/" + getState());
         // showStatus("Имя/состояние флажка: " + getLabel() +
         //            "/" + getState());
            super.processItemEvent(ie);
        }
    }
}
```

Расширение группы флажков

Приведенная ниже программа является модификацией предыдущего примера с флажком. На этот раз группу флажков формируют несколько флажков. Таким образом, в одно и то же время можно установить только один флажок.

```
/*
* <applet code=CheckboxGroupDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class CheckboxGroupDemo2 extends Applet {
    CheckboxGroup cbg;
    MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
    public void init() {
        cbg = new CheckboxGroup();
        myCheckbox1 = new MyCheckbox("Item 1", cbg, true);
        add(myCheckbox1);
        myCheckbox2 = new MyCheckbox("Item 2", cbg, false);
        add(myCheckbox2);
        myCheckbox3 = new MyCheckbox("Item 3", cbg, false);
        add(myCheckbox3);
    }
    class MyCheckbox extends Checkbox {
        public MyCheckbox(String label, CheckboxGroup cbg,
                          boolean flag) {
            super(label, cbg, flag);
```

```
            enableEvents(AWTEvent.ITEM_EVENT_MASK);
        }
        protected void processItemEvent(ItemEvent ie) {
            showStatus("Checkbox name/state: " + getLabel() + "/" +
                       getState());
            super.processItemEvent(ie);
        }
    }
}
```

Расширение класса `Choice`

В следующей программе создается аплет, который отображает список выбора с пунктами, имеющими метки `"Red"`, `"Green"` и `"Blue"`. При выделении пункта в строке состояния средства просмотра аплетов или браузера отображается текст, содержащий название цвета.

Имеется один класс верхнего уровня `ChoiceDemo2`, который расширяет класс `Applet`. Его метод `init()` создает элемент выбора и добавляет его к аплету. Класс `MyChoice` является внутренним классом, который расширяет класс `Choice`. Он вызывает метод `enableEvents()`, чтобы этот объект мог получать извещения о событиях пунктов. Когда происходит событие элемента, вызывается метод `processItemEvent()`. Этот метод отображает текст в строке состояния и вызывает метод `processItemEvent()` для суперкласса.

```
/*
* <applet code=ChoiceDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ChoiceDemo2 extends Applet {
    MyChoice choice;
    public void init() {
        choice = new MyChoice();
        choice.add("Red");
        choice.add("Green");
        choice.add("Blue");
        add(choice);
    }
    class MyChoice extends Choice {
        public MyChoice() {
            enableEvents(AWTEvent.ITEM_EVENT_MASK);
        }
        protected void processItemEvent(ItemEvent ie) {
            showStatus("Choice selection: " + getSelectedItem());
         // showStatus("Состояние выбора: " + getSelectedItem());
            super.processItemEvent(ie);
        }
    }
}
```

Расширение класса `List`

Приведенная ниже программа является модификацией предыдущего примера, но только вместо меню выбора используется список. Существует один класс верхнего уровня `ListDemo2`, который расширяет класс `Applet`. Его метод `init()` создает элемент списка и добавляет его к аплету. Класс `MyList` является внутренним клас-

сом, расширяющим класс List. Он вызывает метод enableEvents(), так что этот объект может получать извещения о событии действия и элемента. Если установить или сбросить отметку с элемента, будет вызван метод processItemEvent(). При двойном щелчке на элементе вызывается также метод processActionEvent(). Оба метода отображают строку, после чего передают управление суперклассу.

```
/*
* <applet code=ListDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ListDemo2 extends Applet {
    MyList list;
    public void init() {
        list = new MyList();
        list.add("Red");
        list.add("Green");
        list.add("Blue");
        add(list);
    }
    class MyList extends List {
        public MyList() {
            enableEvents(AWTEvent.ITEM_EVENT_MASK |
                         AWTEvent.ACTION_EVENT_MASK);
        }
        protected void processActionEvent(ActionEvent ae) {
            showStatus("Action event: " + ae.getActionCommand());
          // showStatus("Событие действия: " + ae.getActionCommand());
            super.processActionEvent(ae);
        }
        protected void processItemEvent(ItemEvent ie) {
            showStatus("Item event: " + getSelectedItem());
            super.processItemEvent(ie);
        }
    }
}
```

Расширение класса Scrollbar

В следующей программе создается аплет, который отображает полосу прокрутки. При работе с этим элементом управления в строке состояния средства просмотра аплетов или браузера отображается строка, которая включает значение, представляемое полосой прокрутки.

Имеется только один класс верхнего уровня ScrollbarDemo2, который расширяет класс Applet. Его метод init() создает полосу прокрутки и добавляет ее к аплету. Класс MyScrollbar является внутренним классом, расширяющим класс Scrollbar. Он вызывает метод enableEvents(), так что этот объект может получать извещения о событиях настройки. При работе с полосой прокрутки вызывается метод processAdjustmentEvent(), а при выборе элемента — метод processAdjustmentEvent(). Он отображает строку, после чего передает управление суперклассу.

```
/*
* <applet code=ScrollbarDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;

public class ScrollbarDemo2 extends Applet {
    MyScrollbar myScrollbar;
    public void init() {
        myScrollbar = new MyScrollbar(Scrollbar.HORIZONTAL,
                                      0, 1, 0, 100);
        myScrollbar.setPreferredSize(new Dimension(100, 20));
        add(myScrollbar);
    }
    class MyScrollbar extends Scrollbar {
        public MyScrollbar(int style, int initial, int thumb,
                           int min, int max) {
            super(style, initial, thumb, min, max);
            enableEvents(AWTEvent.ADJUSTMENT_EVENT_MASK);
        }
        protected void processAdjustmentEvent(AdjustmentEvent ae) {
            showStatus("Adjustment event: " + ae.getValue());
         // showStatus("Событие настройки: " + ae.getValue());
            setValue(getValue());
            super.processAdjustmentEvent(ae);
        }
    }
}
```

Несколько слов о переопределении метода `paint()`

Прежде чем завершить изучение элементов управления библиотеки AWT, скажем несколько слов о переопределении метода `paint()`. Хотя в этой книге не было представлено достаточно простых примеров применения библиотеки AWT с переопределением метода `paint()`, бывают ситуации, когда необходим вызов реализации метода `paint()` суперкласса. Поэтому для некоторых программ вы должны будете использовать следующий шаблон метода `paint()`.

```
public void paint(Graphics g) {

    // код перерисовки данного окна

    // Вызов метода paint() суперкласса
    super.paint(g);
}
```

В языке Java есть два общих типа компонентов: *тяжеловесные* (heavyweight) и *легковесные* (lightweight). У тяжеловесных компонентов есть собственное базовое окно. Легковесный компонент полностью реализуется в коде Java и использует окно, предоставленное родителем. Все элементы управления AWT, описанные и используемые в этой главе, являются тяжеловесными. Но если контейнер содержит какие-нибудь легковесные компоненты (т.е. имеет дочерние легковесные компоненты), ваше переопределение метода `paint()` для этого контейнера должно вызвать метод `super.paint()`. При вызове метода `super.paint()` вы гарантируете правильность перерисовки любых легковесных дочерних компонентов, таких как легковесные элементы управления. Если вы не уверены в типе дочернего компонента, то для выяснения можете вызвать метод `isLightweight()`, определенный в классе `Component`. Он возвращает значение `true`, если компонент легковесный, и значение `false` — в противном случае.

ГЛАВА 26 Изображения

В этой главе будет рассмотрен класс `Image` библиотеки AWT и пакет `java.awt.image`. Вместе они обеспечивают поддержку *воспроизведения изображений*, которое заключается в отображении графических изображений и манипулировании ими. *Изображением* является обычный прямоугольный графический объект. В веб-проектировании изображения являются ключевым компонентом. Когда в 1993 году разработчики из NCSA (National Center for Supercomputer Applications – Национальный центр по применению суперкомпьютеров) решили включить дескриптор `<img>` в браузер Mosaic, это способствовало быстрому развитию системы веб. Этот дескриптор использовался для *встраивания* изображения в поток гипертекста. Язык Java расширяет этот базовый принцип, позволяя программно управлять изображениями. Благодаря этой важной особенности, Java обеспечивает всестороннюю поддержку воспроизведения изображений.

Изображения представляют собой объекты класса `Image`, который является частью пакета `java.awt`. Манипулирование изображениями осуществляется с помощью классов пакета `java.awt.image`, который содержит большое количество классов воспроизведения изображений и интерфейсов. Не имея возможности рассмотреть каждый класс и интерфейс, мы сосредоточимся на тех из них, которые участвуют в основном процессе воспроизведения изображений. Ниже перечислены классы пакета `java.awt.image`, которые будут рассмотрены в этой главе.

`CropImageFilter`	`MemoryImageSource`
`FilteredImageSource`	`PixelGrabber`
`ImageFilter`	`RGBImageFilter`

Мы будем применять следующие интерфейсы.

`ImageConsumer`	`ImageObserver`	`ImageProducer`

Также в главе рассматривается класс `MediaTracker`, который является частью пакета `java.awt`.

Форматы файлов

Первоначально веб-изображения могли быть представлены только в формате GIF. Формат изображений GIF был разработан специалистами из компании CompuServe в 1987 году для обеспечения просмотра изображений в оперативном режиме, поэтому его удобно было использовать и для Интернета. Каждое изображение GIF может содержать не более 256 цветов. В связи с этим ограничением, в 1995 году ведущие разработчики браузеров включили поддержку изображений JPEG. Формат JPEG был разработан группой специалистов-фотографов для хранения псевдополутоновых изображений с полным спектром цветов. Если правиль-

но сформировать подобное изображение, то оно будет передавать более высокую степень точности и его можно будет сжимать более компактно, чем аналогичное изображение в формате GIF. Другим форматом файла является PNG. Это тоже разновидность формата GIF. Как правило, в своих программах вам не придется решать, какой формат нужно использовать. Классы изображений Java абстрагируют все различия между форматами благодаря безупречному интерфейсу.

Основы работы с изображениями: создание, загрузка и отображение

В процессе работы с изображениями вы будете выполнять, в основном, три операции: создание изображения, его загрузка и отображение. Чтобы обратиться к изображениям, которые хранятся в памяти, а также к изображениям, которые необходимо загрузить из внешних источников данных, в языке Java используется класс `Image`. Таким образом, Java обеспечивает способы создания нового объекта изображения и способы его загрузки. Помимо этого, Java предлагает функциональные средства, позволяющие отображать изображения. Все это мы рассмотрим прямо сейчас.

Создание объекта класса `Image`

Могло показаться, что для создания изображения в памяти требуется примерно такая строка.

```
Image test = new Image(200, 100); // Ошибка – не будет работать
```

Однако это не так. Любое изображение предназначено для того, чтобы его можно было отобразить на экране монитора, а класс `Image` не располагает достаточным количеством информации об условиях, необходимых для создания подходящего формата данных для экрана. Поэтому класс `Component` пакета `java.awt` включает метод `createImage()`, используемый для создания объектов класса `Image`. (Имейте в виду, что все компоненты AWT являются подклассами класса `Component`, поэтому каждый из них поддерживает данный метод.)

Метод `createImage()` имеет следующие две формы.

```
Image createImage(ImageProducer производИзображения)
Image createImage(int ширина, int высота)
```

В первом случае возвращается изображение, созданное с помощью производителя (параметр *производИзображения*), который представляет собой объект класса, реализующий интерфейс `ImageProducer`. (О производителях изображений поговорим чуть позже.) Во втором случае возвращается пустое изображение, имеющее определенную ширину и высоту. Ниже показан его пример.

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
```

Здесь для формирования объекта класса `Image` будет создан экземпляр класса `Canvas` и вызван метод `createImage()`. Пока что изображение остается пустым. Позже вы увидите, как можно записать в него данные.

Загрузка изображения

Чтобы получить определенное изображение, его можно также загрузить. Для этого служит метод `getImage()`, определенный в классе `Applet`. Он имеет следующие формы.

```
Image getImage(URL url)
Image getImage(URL url, String имяИзображения)
```

В первом случае будет возвращен объект класса `Image`, который инкапсулирует изображение, хранящееся по адресу, определяемому в параметре *url*. Во втором случае будет возвращен объект класса `Image`, инкапсулирующий изображение, адрес и имя которого определяются параметрами *url* и *имяИзображения* соответственно.

Отображение изображения

После того как получите изображение, его можно будет отобразить с помощью метода `drawImage()`, который является членом класса `Graphics`. Этот метод имеет несколько форм. Ниже представлена форма, которая используется в настоящей книге.

```
boolean drawImage(Image объектИзображения, int слева, int сверху,
                  ImageObserver объектИзображения)
```

В этом случае будет отображено изображение, передаваемое при помощи параметра *объектИзображения*; верхний левый угол изображения определяют параметры *слева* и *сверху*. Параметр *объектИзображения* представляет ссылку на класс, который реализует интерфейс `ImageObserver`. Этот интерфейс реализуют все компоненты библиотек AWT и Swing. *Наблюдатель изображения* представляет собой объект, который может вести наблюдение за изображением во время его загрузки. Интерфейс `ImageObserver` рассмотрим в следующем разделе.

Загрузить и отобразить изображение с помощью методов `getImage()` и `drawInage()` несложно. Ниже показан пример аплета, который загружает и отображает отдельное изображение. В этом аплете загружается файл `seattle.jpg`, однако для данного примера можете взять любой другой файл формата GIF, JPG или PNG (при условии, что он будет находиться в том же каталоге, что и файл HTML, содержащий аплет).

```
/*
* <applet code="SimpleImageLoad" width=248 height=146>
* <param name="img" value="seattle.jpg">
* </applet>
*/
import java.awt.*;
import java.applet.*;

public class SimpleImageLoad extends Applet
{
    Image img;
    public void init() {
        img = getImage(getDocumentBase(), getParameter("img"));
    }

    public void paint(Graphics g) {
        g.drawImage(img, 0, 0, this);
    }
}
```

В методе `init()` переменной `img` присваивается изображение, возвращенное методом `getImage()`. Метод `getImage()` использует строку, возвращенную методом `getParameter("img")`, в качестве имени файла с изображением. Это изображение загружается из URL, который связан с результатом выполнения метода `getDocumentBase()`, т.е. URL страницы HTML, в которой находился этот

дескриптор аплета. Имя файла, возвращенное методом `getParameter("img")`, получено из дескриптора аплета `<param name="img" value="seattle.jpg">`. Такая схема эквивалентна применению дескриптора HTML `<img scr="seattle.jpg" width=248 height=146>`, хотя она работает немного медленнее. На рис. 26.1 можно видеть результат выполнения этой программы.

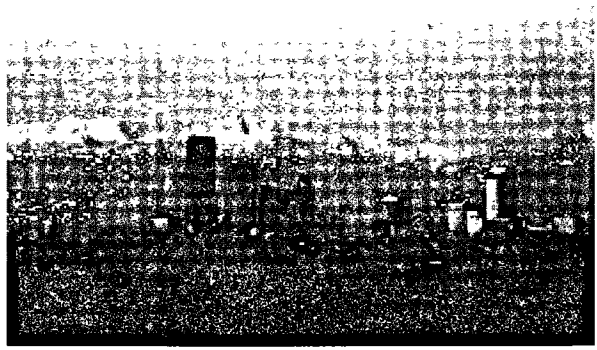


Рис. 26.1. Результат выполнения программы `SimpleImageLoad`

Выполнение этого аплета начинается с загрузки изображения `img` в методе `init()`. На экране монитора вы будете видеть изображение в процессе его загрузки из сети, поскольку при реализации интерфейса `ImageObserver` классом `Applet` метод `paint()` вызывается каждый раз при поступлении новых данных, связанных с изображением.

Конечно, неплохо иметь возможность наблюдать за ходом загрузки изображения, однако будет лучше, если время, отводимое на загрузку изображения, вы потратите на параллельное выполнение других задач. Это позволит отобразить полностью сформированное изображение на экране монитора мгновенно сразу же после его загрузки. Интерфейс `ImageObserver`, о котором пойдет речь далее, можно применять для наблюдения за ходом загрузки изображения в момент вывода на экран монитора какой-то другой информации.

Интерфейс `ImageObserver`

Описываемый интерфейс используется для получения уведомления о формировании изображения; он определяет только один метод — `imageUpdate()`. Используя наблюдатель изображений, вы сможете выполнять ряд других действий — отображать индикатор процесса выполнения или, к примеру, переключаться на экран при получении уведомления о ходе процесса загрузки. Этот тип уведомлений полезен при загрузке изображения по низкоскоростной сети.

Метод `imageUpdate()` имеет следующую общую форму.

```
boolean imageUpdate(Image объектИзображения, int флаги, int слева,
                    int сверху, int ширина, int высота)
```

Здесь параметр *объектИзображения* представляет загружаемое изображение, а параметр *флаги* — целое число, которое показывает состояние обновляемого отчета. Четыре целочисленных параметра, *слева*, *сверху*, *ширина* и *высота*, определяют прямоугольник, значения которого зависят от значений, передаваемых в параметре *флаги*. Метод `imageUpdate()` возвращает значение `false`, если

процесс загрузки был завершен, и значение `true` – если необходимо обработать еще одно изображение.

Параметр *флаги* содержит один или несколько битовых флагов, определяемых как статические переменные в рамках интерфейса `ImageObserver`. Эти флаги, а также передаваемая с их помощью информация представлены в табл. 26.1.

Таблица 26.1. Битовые флаги параметра `флаги` метода `imageUpdate()`

Флаг	Назначение
`WIDTH`	Параметр *ширина* является действительным и содержит значение ширины изображения
`HEIGHT`	Параметр *высота* является действительным и содержит значение высоты изображения
`PROPERTIES`	Свойства, связанные с изображением, можно получить с помощью метода `imgObj.getProperty()`
`SOMEBITS`	Были получены дополнительные пиксели, необходимые для рисования изображения. Параметры *слева*, *сверху*, *ширина* и *высота* определяют прямоугольник, содержащий новые пиксели
`FRAMEBITS`	Был получен весь кадр, являющийся частью ранее нарисованного многокадрового изображения. Этот кадр можно отобразить. Параметры *слева*, *сверху*, *ширина* и *высота* не используются
`ALLBITS`	Изображение готово. Параметры *слева*, *сверху*, *ширина* и *высота* не используются
`ERROR`	Обнаружена ошибка в изображении, за которым велось асинхронное наблюдение. Изображение не готово и не может быть отображено. Не получено никакой дополнительной информации об изображении. Будет установлен также флаг `ABORT`, чтобы показать, что процесс формирования изображения был прерван
`ABORT`	Формирование изображения, за которым велось асинхронное наблюдение, было прервано до того, как оно было полностью готово. Однако если не было ошибки, то при попытке обращения к какой-либо части данных изображения начнется повторное формирование изображения

Класс `Applet` имеет реализацию метода `imageUpdate()` согласно интерфейсу `ImageObserver`, который используется для перерисовки изображений во время их загрузки. Вы можете изменить это поведение, если переопределите данный метод в своем классе.

Ниже показан пример применения метода `imageUpdate()`.

```
public boolean imageUpdate(Image img, int flags,
                           int x, int y, int w, int h) {
    if ((flags & ALLBITS) == 0) {
        System.out.println("Still processing the image.");
     // System.out.println("Изображение все еще обрабатывается.");
        return true;
    } else {
        System.out.println("Done processing the image.");
     // System.out.println("Обработка изображения завершена.");
        return false;
    }
}
```

Двойная буферизация

Изображения можно использовать не только для хранения картинок и рисунков, как было показано только что, но и в качестве внеэкранных поверхностей рисования. С их помощью можно визуализировать любое изображение, включая текст и графику во внеэкранном буфере, содержимое которого можно будет отобразить через некоторый промежуток времени. Преимущество такого подхода заключается в том, что изображение можно будет увидеть лишь после того, как оно будет полностью готово. Для рисования сложной поверхности может потребоваться несколько миллисекунд или более, причем для пользователя этот процесс может выглядеть как серия вспышек или мерцаний.

Подобные эффекты отвлекают внимание и приводят к тому, что пользователь воспринимает визуализируемое изображение гораздо медленнее, чем это происходит на самом деле. Процесс использования внеэкранного изображения для уменьшения мерцания называется *двойной буферизацией*, поскольку экран монитора принимается в качестве буфера для пикселей, а внеэкранное изображение является вторым буфером, в котором можно подготавливать пиксели для визуализации.

Вы уже видели в этой главе, как создается пустой объект класса `Image`. Сейчас будет показано, как осуществляется рисование изображения на самом экране. Если вы помните из предыдущих глав, для этой цели нужен объект класса `Graphics`, благодаря которому можно будет использовать любые методы визуализации, доступные в Java. Он удобен тем, что доступ к объекту класса `Graphics`, который можно применять для рисования изображения, осуществляется с помощью метода `getGraphics()`. Ниже показан фрагмент кода, в котором создается новое изображение, принимается графическое содержимое и все изображение заполняется красными пикселями.

```
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 того как создадите и заполните внеэкранное изображение, его видно не будет. Чтобы отобразить искомое изображение, вызовите метод `drawImage()`. Ниже показан пример, в котором для рисования изображения требуется много времени. Вы сможете сравнить, как двойная буферизация влияет на восприятие времени рисования.

```
/*
* <applet code=DoubleBuffer width=250 height=250>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class DoubleBuffer extends Applet {
    int gap = 3;
    int mx, my;
    boolean flicker = true;
    Image buffer = null;
    int w, h;

    public void init() {
        Dimension d = getSize();
        w = d.width;
        h = d.height;
```

```
        buffer = createImage(w, h);
        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();
            }
        });
    }
    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);
     // g.drawString("Щелкните для включения двойной буферизации",
                     10, h/2);

        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);
    }
}
```

Этот простой аплет имеет сложный метод `paint()`. Он окрашивает фон в синий цвет, а затем рисует поверх него красный муар. Вверху он выводит текст черного цвета и вычерчивает желтую окружность, центр которой определяется по координатам `mx`, `my`. Методы `mouseMoved()` и `mouseDragged()` заменяются для отслеживания положения курсора мыши. Эти методы идентичны друг другу, за исключением булевой переменной `flicker`. Метод `mouseMoved()` присваивает переменной `flicker` значение `true`, а метод `mouseDragger()` — значение `false`. Это равнозначно эффекту вызова метода `repaint()`, когда переменная `flicker` имеет значение `true` при перемещении курсора мыши (без щелчка ее кнопкой) и значение `false` — при перемещении курсора мыши с удерживанием кнопки в нажатом состоянии.

Если метод `paint()` вызывается тогда, когда переменная `flicker` имеет значение `true`, мы будем видеть на экране монитора выполнение каждой операции рисования. Если был произведен щелчок кнопкой мыши и метод `paint()` вызывается, если переменная `flicker` имеет значение `false`, будем наблюдать несколько иную картину. Метод `paint()` заменяет ссылку `g` на класс `Graphics` графическим содержимым внеэкранного холста, `buffer`, который мы создали с помощью метода `init()`. Затем все операции рисования становятся невидимыми. В завершающей части работы метода `paint()` мы просто вызываем метод `drawImage()` для одновременного отображения результатов всех методов рисования.

Обратите внимание на то, что теперь можно передавать методу `drawImage()` значение `null` в четвертом параметре. Этот параметр служит для передачи объекта `ImageObserver`, который получает извещение о событиях изображения. Поскольку это изображение не формируется из сетевого потока, уведомления не нужны. Левый снимок экрана на рис. 26.2 показывает, как будет выглядеть аплет, если не будет произведен щелчок кнопкой мыши. Как можно заметить, снимок был получен как раз в тот момент, когда изображение было наполовину перерисованным. Правый снимок показывает, что при нажатой кнопке мыши изображение оказывается полностью сформированным благодаря использованию двойной буферизации.

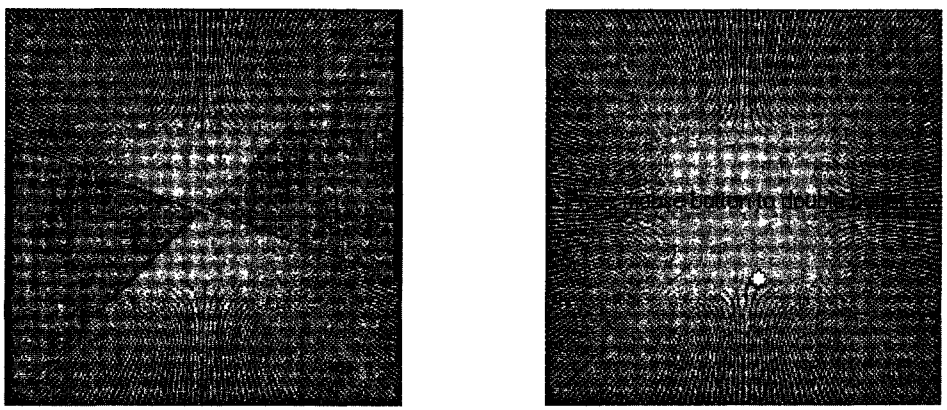


Рис. 26.2. Результат выполнения аплета `DoubleBuffer` без использования двойной буферизации (слева) и с использованием двойной буферизации (справа)

Класс `MediaTracker`

Объект класса `MediaTracker` представляет собой объект, который параллельно проверяет состояние произвольного количества изображений. Чтобы использовать класс `MediaTracker`, нужно создать его новый экземпляр и использовать его метод `addImage()` для наблюдения за состоянием загрузки изображения. Метод `addImage()` имеет следующие общие формы.

```
void addImage(Image объектИзображения, int идентификаторИзображения)
void addImage(Image объектИзображения, int идентификаторИзображения, int
ширина, int высота)
```

Здесь параметр *объектИзображения* представляет отслеживаемое изображение. Его идентификатор передается в параметре *идентификаторИзображения*. Идентификаторы не обязательно должны быть уникальными. Один и тот же идентификатор можно использовать в нескольких изображениях, обозначая их как часть группы. Кроме того, изображения с более низкими идентификаторами при

загрузке имеют приоритет перед таковыми с более высокими идентификаторами. Во второй форме параметры *ширина* и *высота* определяют размеры объекта при его отображении.

После того как изображение будет зарегистрировано, можно проверить, загружено ли оно, или подождать, пока оно полностью не загрузится. Чтобы проверить состояние изображения, вызовите метод `checkID()`. В этой главе используется следующий вариант этого метода.

```
boolean checkID(int идентификаторИзображения)
```

Здесь параметр *идентификаторИзображения* определяет идентификационный номер изображения, которое вы хотите проверить. Метод возвращает значение `true`, если были загружены все изображения, имеющие заданный идентификатор (или если процесс загрузки был остановлен вследствие ошибки или прерван пользователем). В противном случае он возвращает значение `false`. Вы можете использовать метод `checkAll()` для просмотра, все ли наблюдаемые изображения были загружены.

Класс `MediaTracker` следует использовать при загрузке группы изображений. Если все интересующие вас изображения еще не загружены, можете отобразить что-нибудь, чтобы отвлечь пользователя, пока не будут полностью загружены все изображения.

Внимание! Если объект класса `MediaTracker` использовать после вызова метода `addImage()`, то ссылка на класс `MediaTracker` предотвратит процесс сбора "мусора" в системе. Если вы хотите, чтобы система могла запускать сборщик "мусора" относительно отслеживаемых изображений, убедитесь в том, что он будет выполняться и в отношении экземпляра класса `MediaTracker`.

Ниже показан пример, в котором загружается семь изображений и отображается привлекательная диаграмма хода выполнения загрузки.

```
/*
/*
* <applet code="TrackedImageLoad" width=300 height=400>
* <param name="img"
* value="vincent+leonardo+matisse+picasso+renoir+seurat+vermeer">
* </applet>
*/
import java.util.*;
import java.applet.*;
import java.awt.*;

public class TrackedImageLoad extends Applet implements Runnable {
    MediaTracker tracker;
    int tracked;
    int frame_rate = 5;
    int current_img = 0;
    Thread motor;
    static final int MAXIMAGES = 10;
    Image img[] = new Image[MAXIMAGES];
    String name[] = new String[MAXIMAGES];
    volatile boolean stopFlag;

    public void init() {
        tracker = new MediaTracker(this);
        StringTokenizer st = new StringTokenizer(getParameter("img"),
                                                 "+");

        while(st.hasMoreTokens() && tracked <= MAXIMAGES) {
            name[tracked] = st.nextToken();
```

```
            img[tracked] = getImage(getDocumentBase(),
                                    name[tracked] + ".jpg");
            tracker.addImage(img[tracked], tracked);
            tracked++;
        }
    }

    public void paint(Graphics g) {
        String loaded = "";
        int donecount = 0;

        for(int i=0; i<tracked; i++) {
            if (tracker.checkID(i, true)) {
                donecount++;
                loaded += name[i] + " ";
            }
        }

        Dimension d = getSize();
        int w = d.width;
        int h = d.height;

        if (donecount == tracked) {
            frame_rate = 1;
            Image i = img[current_img++];
            int iw = i.getWidth(null);
            int ih = i.getHeight(null);
            g.drawImage(i, (w - iw)/2, (h - ih)/2, null);
            if (current_img >= tracked)
                current_img = 0;
        } else {
            int x = w * donecount / tracked;
            g.setColor(Color.black);
            g.fillRect(0, h/3, x, 16);
            g.setColor(Color.white);
            g.fillRect(x, h/3, w-x, 16);
            g.setColor(Color.black);
            g.drawString(loaded, 10, h/2);
        }
    }

    public void start() {
        motor = new Thread(this);
        stopFlag = false;
        motor.start();
    }

    public void stop() {
        stopFlag = true;
    }

    public void run() {
        motor.setPriority(Thread.MIN_PRIORITY);
        while (true) {
            repaint();
            try {
                Thread.sleep(1000/frame_rate);
            } catch (InterruptedException e) {
                System.out.println("Interrupted");
                return;
            }
```

```
            if(stopFlag)
                return;
        }
    }
}
```

В этом примере создается новый экземпляр класса `MediaTracker` в методе `init()`, после чего с помощью метода `addImage()` каждое из указанных изображений добавляется как отслеженное. В методе `paint()` вызывается метод `checkID()` для каждого изображения, за которым велось наблюдение. После загрузки все изображения будут выведены на экран. В противном случае отображается простая диаграмма, информирующая о количестве загруженных изображений, а под ней выводятся имена полностью загруженных изображений. На рис. 26.3 показаны две сцены из этого выполняющегося аплета. На одной из них видна диаграмма, информирующая о загрузке трех изображений. Другая сцена — это автопортрет Ван Гога во время демонстрации слайда.

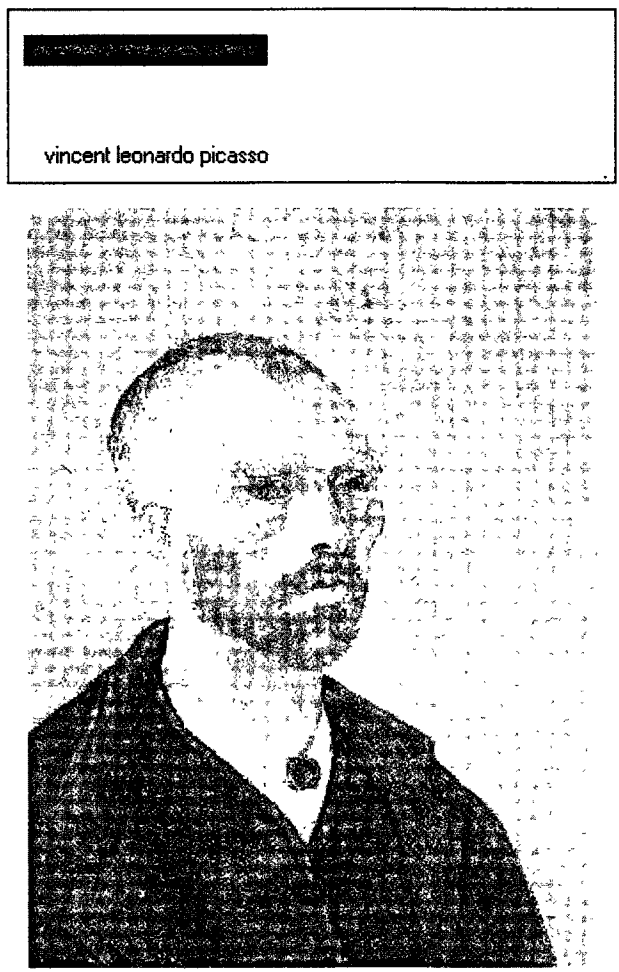


Рис. 26.3. Аплет `TrackedImageLoad` во время выполнения

Интерфейс ImageProducer

Интерфейс ImageProducer является интерфейсом для объектов, которые должны подготовить данные для изображений. Объект, реализующий интерфейс ImageProducer, задает целочисленный или байтовый массив, представляющий данные изображений, и формирует объекты класса Image. Как вы могли видеть ранее, одна из форм метода createImage() получает объект интерфейса ImageProducer в качестве своего параметра. Пакет java.awt.image содержит два производителя изображений — классы MemoryImageSource и FilteredImageSource. Здесь мы рассмотрим класс MemoryImageSource и создадим новый объект класса Image на основе данных, созданных в аплете.

Класс MemoryImageSource

Этот класс формирует новое изображение класса Image на основе массива данных. Он определяет несколько конструкторов. Ниже показан один из конструкторов, который мы будем использовать.

```
MemoryImageSource(int ширина, int высота, int пиксель[],
                  int смещение, int ширинаСтрокиРазвертки)
```

Объект класса MemoryImageSource формируется на основе массива целых чисел *пиксель* в цветовой модели RGB, которая используется по умолчанию для подготовки данных для объекта класса Image. В цветовой модели, используемой по умолчанию, пиксель представляет собой целое число со значениями альфа, красной, зеленой и синей составляющих (0xAARRGGBB). Значение альфа представляет степень прозрачности пикселя. Полностью прозрачному пикселю соответствует нулевое значение, а полностью непрозрачному — значение 255. Значения ширины и высоты готового изображения передаются в параметрах *ширина* и *высота*. Начальная точка в массиве пикселей, с которой начнется чтение данных, определяется параметром *смещение*. Ширина строки развертки (которая часто определяет то же, что и ширина изображения) указывается параметром *ширинаСтрокиРазвертки*.

В следующем коротком примере создается объект класса MemoryImageSource с помощью разновидности простого алгоритма (побитовое исключающее "ИЛИ" координат x и y каждого пикселя), взятого из книги *Beyond Photography, The Digital Darkroom* Джерарда Дж. Хольцманна (Gerard J. Holzmann, Prentice Hall, 1988).

```
/*
* <applet code="MemoryImageGenerator" width=256 height=256>
* </applet>
*/
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class MemoryImageGenerator extends Applet {
    Image img;
    public void init() {
        Dimension d = getSize();
        int w = d.width;
        int h = d.height;
        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));
    }

    public void paint(Graphics g) {
        g.drawImage(img, 0, 0, this);
    }
}
```

Данные для нового объекта класса `MemoryImageSource` создаются в методе `init()`. Массив целых чисел предназначен для хранения значений пикселей; данные создаются во вложенных циклах `for`, в которых значения `r`, `g` и `b` получаются смещенными на пиксель в массиве `pixels`. В конце вызывается метод `createImage()` с новым экземпляром класса `MemoryImageSource`, созданным из базовых данных о пикселях в качестве его параметра. На рис. 26.4 показано изображение в момент запуска аплета. (В цвете оно выглядит гораздо привлекательнее.)

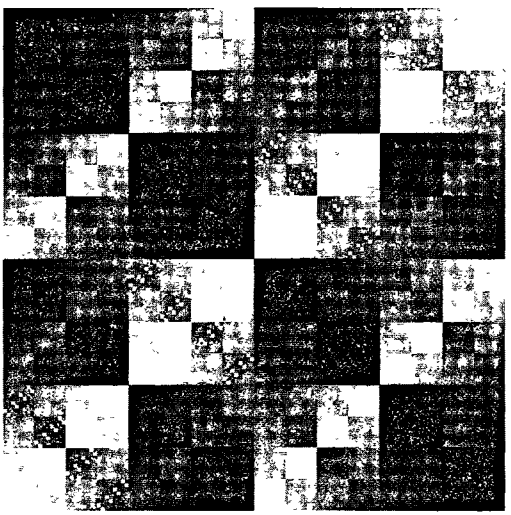


Рис. 26.4. Результат выполнения аплета `MemoryImageGenerator`

Интерфейс `ImageConsumer`

Интерфейс `ImageConsumer` является интерфейсом для объектов, которые будут получать данные о пикселях из изображений и задавать их в качестве другой разновидности данных. Таким образом, этот интерфейс является прямой противоположностью описанному ранее интерфейсу `ImageProducer`. Объект, реализующий интерфейс `ImageConsumer`, будет создавать массивы типа `int` или `byte`, представляющие пиксели из объекта класса `Image`. Мы рассмотрим класс `PixelGrabber`, который является простой реализацией интерфейса `ImageConsumer`.

Класс PixelGrabber

В пакете java.lang.image определен класс PixelGrabber, который является прямой противоположностью классу MemoryImageSource. Вместо того чтобы формировать изображение на основе массива значений пикселей, он принимает существующее изображение и *захватывает* в нем массив пикселей. Чтобы использовать класс PixelGrabber, вы сначала создаете массив целых чисел, размер которого позволяет хранить данные о пикселях, после чего создаете экземпляр класса PixelGrabber, передавая прямоугольную область, которую необходимо захватить. Затем для этого экземпляра нужно вызывать метод grabPixels().

Ниже показан конструктор класса PixelGrabber, используемый в этой главе.

```
PixelGrabber(Image объектИзображения, int слева, int сверху,
             int ширина, int высота, int пиксель[ ], int смещение,
             int ширинаСтрокиРазвертки)
```

Здесь параметр *объектИзображения* представляет объект, пиксели которого будут захвачены. Значения параметров *слева* и *сверху* указывают верхний левый угол прямоугольной области, а значения параметров *ширина* и *высота* задают размеры прямоугольной области, из которой будут получены пиксели. Пиксели будут храниться в массиве *пиксель*, начиная со смещения, определенного в параметре *смещение*. Ширина строки развертки (часто равнозначна ширине изображения) передается при помощи соответствующего параметра (*ширинаСтрокиРазвертки*).

Метод grabPixels() определяется следующим образом.

```
boolean grabPixels() throws InterruptedException
boolean grabPixels(long миллисекунды) throws InterruptedException
```

Оба метода возвращают значение true при успешном завершении выполнения и значение false — в противном случае. Во второй форме метода параметр *миллисекунды* определяет время, в течение которого метод будет ожидать получение пикселей. Оба метода создают исключение InterruptedException, если выполнение прерывается другим потоком.

Ниже показан пример, в котором производится захват пикселей в изображении с последующим построением гистограммы яркости пикселей. *Гистограмма* представляет собой простой подсчет пикселей, имеющих некоторую яркость, для всех значений яркости между 0 и 255. После того как аплет нарисует изображение, поверх него выводится гистограмма.

```
/*
* <applet code=HistoGrab.class width=341 height=400>
* <param name=img value=vermeer.jpg>
* </applet> */
import java.applet.*;
import java.awt.* ;
import java.awt.image.* ;

public class HistoGrab extends Applet {
    Dimension d;
    Image img;
    int iw, ih;
    int pixels[];
    int w, h;
    int hist[] = new int[256];
    int max_hist = 0;

    public void init() {
        d = getSize();
        w = d.width;
```

```
        h = d.height;

        try {
            img = getImage(getDocumentBase(), getParameter("img"));
            MediaTracker t = new MediaTracker(this);

            t.addImage(img, 0);
            t.waitForID(0);
            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;
        }

        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];
        }
    }

    public void update() {}

    public void paint(Graphics g) {
        g.drawImage(img, 0, 0, null);
        int x = (w - 256) / 2;
        int lasty = h - h * hist[0] / max_hist;

        for (int i=0; i<256; i++, x++) {
            int y = h - h * hist[i] / max_hist;
            g.setColor(new Color(i, i, i));
            g.fillRect(x, y, 1, h);
            g.setColor(Color.red);
            g.drawLine(x-1,lasty,x,y);
            lasty = y;
        }
    }
}
```

На рис. 26.5 показано изображение и гистограмма знаменитой картины Вермеера (Vermeer).

Класс `ImageFilter`

При наличии пары интерфейсов `ImageProducer` и `ImageConsumer`, а также их классов `MemoryImageSource` и `PixelGrabber` можно создать произвольный набор фильтров преобразования, которые будут принимать источники пикселей, модифицировать их и передавать некоторому потребителю. Этот механизм анало-

гичен механизму создания определенных классов на основе абстрактных классов ввода-вывода `InputStream`, `OutputStream`, `Reader` и `Writer` (они рассматривались в главе 19). Данная потоковая модель для изображений завершается за счет введения класса `ImageFilter`.

Рис. 26.5. Результат выполнения аплета `HistoGrab`

Пакет `java.awt.image` содержит подклассы класса `ImageFilter`, к которым относятся классы `AreaAveragingScaleFilter`, `CropImageFilter`, `ReplicateScaleFilter` и `RGBImageFilter`. Существует также реализация интерфейса `ImageProducer` в виде класса `FilteredImageSource`, который принимает произвольный класс `ImageFilter` и "вкладывает в него" интерфейс `ImageProducer` для фильтрации формируемых им пикселей. Экземпляр класса `FilteredImageSource` можно использовать в качестве экземпляра интерфейса `ImageProducer` при вызовах метода `createImage()` почти так же, как и класс `BufferedInputStream` можно применять в качестве класса `InputStream`.

В этой главе рассмотрим два класса фильтра — `CropImageFilter` и `RGBImageFilter`.

Фильтр класса `CropImageFilter`

Фильтр класса `CropImageFilter` осуществляет фильтрацию источника изображения для извлечения прямоугольной области. Этот фильтр будет полезен, напри-

мер, при работе с несколькими небольшими изображениями, созданными из одного крупного исходного изображения. Для загрузки двадцати изображений размером 2 Кбайт потребуется больше времени, чем для загрузки одного изображения размером 40 Кбайт, имеющего множество внедренных анимационных кадров. Если каждое составное изображение имеет один и тот же размер, то вы без особого труда сможете извлечь их с помощью фильтра класса `CropImageFilter`, расчленяя весь блок в самом начале работы программы. Ниже показан пример формирования 16 изображений на основе одного большого изображения. Мозаичные элементы затем перетасовываются 32 раза заменой случайной пары, взятой из 16 изображений.

```
/*
* <applet code=TileImage.class width=288 height=399>
* <param name=img value=picasso.jpg>
* </applet>
*/
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class TileImage extends Applet {
    Image img;
    Image cell[] = new Image[4*4];
    int iw, ih;
    int tw, th;

    public void init() {
        try {
            img = getImage(getDocumentBase(), getParameter("img"));
            MediaTracker t = new MediaTracker(this);
            t.addImage(img, 0);
            t.waitForID(0);
            iw = img.getWidth(null);
            ih = img.getHeight(null);
            tw = iw / 4;
            th = ih / 4;
            CropImageFilter f;
            FilteredImageSource fis;
            t = new MediaTracker(this);
            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);
                    t.addImage(cell[i], 1);
                }
            }
            t.waitForAll();
            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 (InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
```

```
    public void update(Graphics g) {
        paint(g);
    }

    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, y * th, null);
            }
        }
    }
}
```

На рис. 26.6 показана известная картина Пикассо, элементы которой перетасованы с помощью аплета `TileImage`.

Рис. 26.6. Результат выполнения аплета `TileImage`

Фильтр класса `RGBImageFilter`

Фильтр класса `RGBImageFilter` используется для преобразования одного изображения в другое, пиксель за пикселем, осуществляя при этом преобразование цветов. Этот фильтр можно использовать для того, чтобы осветлить изображе-

ние, увеличить его контрастность или даже для преобразования его в полутоновое изображение.

Чтобы продемонстрировать работу фильтра класса RGBImageFilter, мы подготовили более сложный пример, который демонстрирует применение динамической стратегии встраивания для фильтров обработки изображений. Мы создали интерфейс для обобщенной фильтрации изображения, чтобы аплет мог просто загружать эти фильтры на основании дескрипторов <param>, не имея при этом предварительной информации о каждом фильтре класса ImageFilter. Пример включает главный класс аплета ImageFilterDemo, интерфейс PlugInFilter и служебный класс LoadedImage, который инкапсулирует некоторые методы класса MediaTracker, используемые в этой главе. Кроме того, используются три класса фильтра, Grayscale, Invert и Contrast, которые просто манипулируют пространством цветов исходного изображения с помощью фильтров класса RGBImageFilter, и два дополнительных класса, Blur и Sharpen, позволяющие реализовать более сложные фильтры свертки, изменяющие данные о пикселях на основе пикселей, окружающих их в исходных данных. Классы Blur и Sharpen являются подклассами абстрактного вспомогательного класса Convolver. Давайте рассмотрим каждую часть нашего примера.

Класс ImageFilterDemo

Этот класс является каркасом аплета для наших примеров применения фильтров изображений. Он использует диспетчер класса BorderLayout и имеет панель класса Panel в позиции *South* для хранения кнопок, которые будут представлять каждый фильтр. Объект Label занимает позицию *North* для информационных сообщений о ходе работы фильтра. В позиции *Center* помещается изображение (которое инкапсулируется в подклассе LoadedImage класса Canvas, о котором упоминалось ранее). Кнопки/фильтры дескриптора filters<param>, разделенные с помощью символа +, анализируются с применением класса StringTokenizer.

Метод actionPerformed() интересен тем, что он использует метку кнопки в качестве имени класса фильтра, который он пытается загрузить с помощью метода (PlugInFilter)Class.forName(a).newInstance(). Этот метод является надежным и выполняет определенное действие, если кнопка не соответствует подходящему классу, реализующему интерфейс PlugInFilter.

```
/*
* <applet code=ImageFilterDemo width=350 height=450>
* <param name=img value=vincent.jpg>
* <param name=filters value="Grayscale+Invert+Contrast+Blur+ Sharpen">
* </applet>
*/
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class ImageFilterDemo extends Applet implements ActionListener {
    Image img;
    PlugInFilter pif;
    Image fimg;
    Image curImg;
    LoadedImage lim;
    Label lab;
    Button reset;

    public void init() {
```

```
        setLayout(new BorderLayout());
        Panel p = new Panel();
        add(p, BorderLayout.SOUTH);
        reset = new Button("Reset");
        reset.addActionListener(this);
        p.add(reset);
        StringTokenizer st = new StringTokenizer(getParameter("filters"),
                                                 "+");
        while(st.hasMoreTokens()) {
            Button b = new Button(st.nextToken());
            b.addActionListener(this);
            p.add(b);
        }
        lab = new Label("");
        add(lab, BorderLayout.NORTH);
        img = getImage(getDocumentBase(), getParameter("img"));
        lim = new LoadedImage(img);
        add(lim, BorderLayout.CENTER);
    }

    public void actionPerformed(ActionEvent ae) {
        String a = "";
        try {
            a = ae.getActionCommand();
            if (a.equals("Reset")) {
                lim.set(img);
                lab.setText("Normal");
            }
            else {
                pif = (PlugInFilter) Class.forName(a).newInstance();
                fimg = pif.filter(this, img);
                lim.set(fimg);
                lab.setText("Filtered: " + a);
            }
            repaint();
        } catch (ClassNotFoundException e) {
            lab.setText(a + " not found");
            lim.set(img);
            repaint();
        } catch (InstantiationException e) {
            lab.setText("couldn't new " + a);
        } catch (IllegalAccessException e) {
            lab.setText("no access: " + a);
        }
    }
}
```

На рис. 26.7 показано, как выглядит аплет, когда он впервые загружается с использованием дескриптора аплета, показанного в начале исходного файла.

Интерфейс `PlugInFilter`

Для абстрактной фильтрации изображений используется интерфейс `PlugInFilter`. Он имеет только один метод, `filter()`, который принимает аплет и исходное изображение и возвращает новое отфильтрованное изображение.

```
interface PlugInFilter {java.awt.Image filter(java.applet.Applet a,
java.awt.Image in);}
```

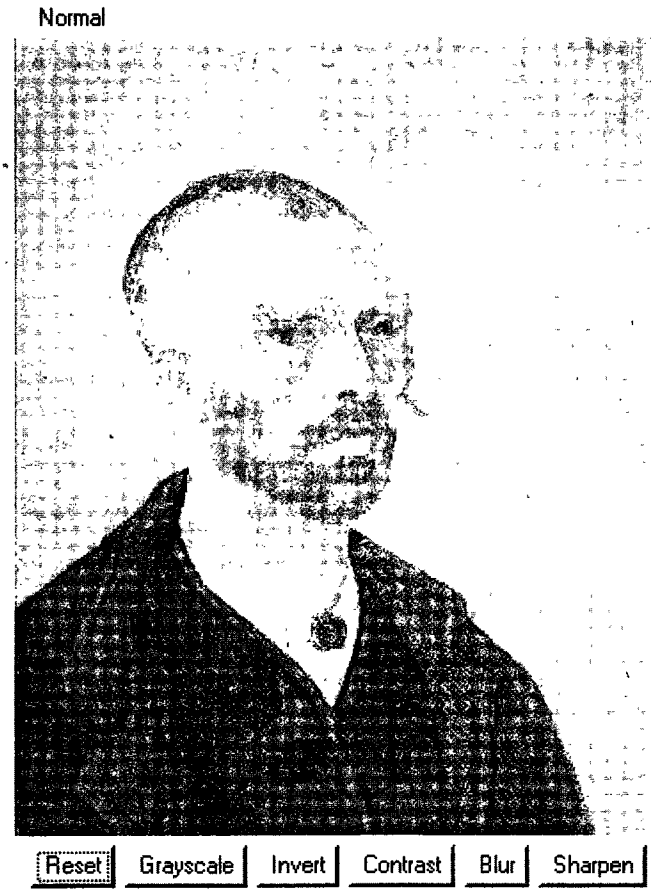


Рис. 26.7. Обычный вывод аплета `ImageFilterDemo`

Класс `LoadedImage`

Вспомогательным классом класса `Canvas`, который принимает изображение во время его формирования и синхронно загружает его с помощью класса `MediaTracker`, является класс `LoadedImage`. В результате этот класс будет вести себя корректно внутри элемента управления `LayoutControl`, поскольку в нем переопределены методы `getPreferredSize()` и `getMinimumSize()`. Также он имеет метод `set()`, служащий для определения нового изображения класса `Image`, которое необходимо отобразить на данном холсте класса `Canvas`. Так отфильтрованное изображение отображается после завершения вставки.

```
import java.awt.*;

public class LoadedImage extends Canvas {
    Image img;

    public LoadedImage(Image i) {
        set(i);
    }
```

```
    void set(Image i) {
        MediaTracker mt = new MediaTracker(this);
        mt.addImage(i, 0);
        try {
            mt.waitForAll();
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
            return;
        }
        img = i;
        repaint();
    }

    public void paint(Graphics g) {
        if (img == null) {
            g.drawString("no image", 10, 30);
        } else {
            g.drawImage(img, 0, 0, this);
        }
    }

    public Dimension getPreferredSize() {
        return new Dimension(img.getWidth(this), img.getHeight(this));
    }

    public Dimension getMinimumSize() {
        return getPreferredSize();
    }
}
```

Класс `Grayscale`

Фильтр класса `Grayscale` является подклассом класса `RGBImageFilter`. Это значит, что объект класса `Grayscale` может использоваться в качестве параметра класса `ImageFilter` для конструктора класса `FilteredImageSource`. Единственное, что необходимо будет сделать, — это переопределить метод `filterRGB()`, чтобы изменить входные значения цветовых составляющих. Он принимает значения красной, зеленой и синей составляющих и вычисляет яркость пикселя с помощью коэффициента преобразования цвета в яркость, утвержденного комитетом NTSC (National Television Standards Committee — Национальный комитет по телевизионным стандартам). Затем он просто возвращает серый пиксель, имеющий ту же яркость, что и источник цвета.

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class Grayscale extends RGBImageFilter implements PlugInFilter {
    public Image filter(Applet a, Image in) {
        return a.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);
    }
}
```

Класс `Invert`

Фильтр класса `Invert` также является достаточно простым. Он принимает отдельно красный, зеленый и синий каналы, а затем инвертирует их, вычитая из них значения 255. Инвертированные значения переносятся обратно в значение пикселя и возвращаются (рис. 26.8).

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
class Invert extends RGBImageFilter implements PlugInFilter {
    public Image filter(Applet a, Image in) {
        return a.createImage(new FilteredImageSource(in.getSource(),
                             this));
    }
    ublic 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);
    }
}
```

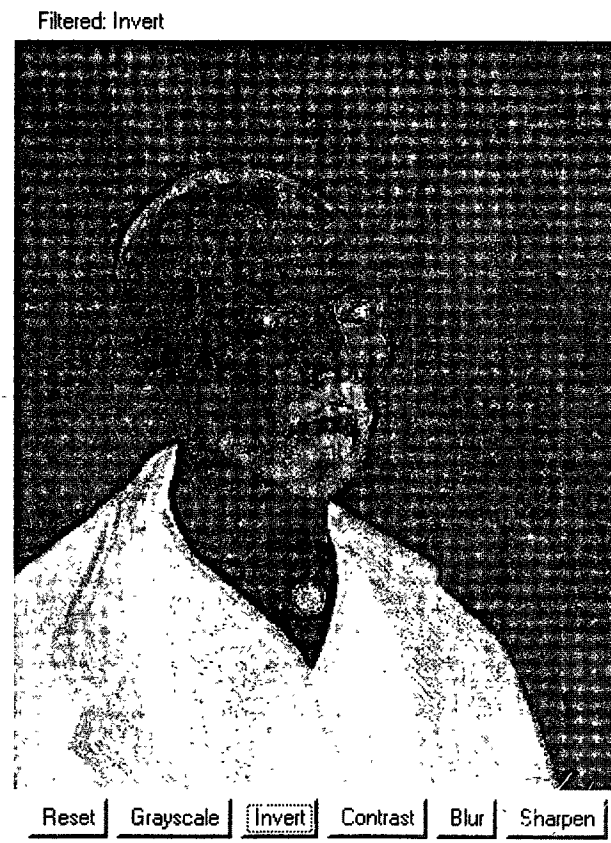


Рис. 26.8. Использование фильтра класса `Invert` в аплете `ImageFilterDemo`

Класс Contrast

Фильтр класса Contrast очень похож на фильтр класса Grayscale, за исключением того, что замена метода filterRGB() производится несколько сложнее. В алгоритме, который он применяет для улучшения контрастности, принимаются значения отдельно для красной, зеленой и синей составляющих и увеличиваются в 1,2 раза, если их яркость выше 128. Если их яркость ниже 128, они делятся на 1,2. Увеличенные значения фиксируются в значении 255 с помощью метода multiclamp().

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class Contrast extends RGBImageFilter implements PlugInFilter {

    public Image filter(Applet a, Image in) {
        return a.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. 26.9 показано изображение после щелчка на кнопке Contrast (Контрастность).

Класс Convolver

Абстрактный класс Convolver обрабатывает основные операции фильтра свертки, реализуя интерфейс ImageConsumer для переноса исходных пикселей в массив imgpixels. Для отфильтрованных данных он создает второй массив newimgpixels. Фильтры свертки выбирают небольшие прямоугольные области пикселей вокруг каждого пикселя в изображении, которые называются *ядром свертки*. Эта область, которая в данном примере имеет размер 3×3, используется для принятия решения, каким образом будет изменяться центральный пиксель в области.

На заметку! Фильтр не может модифицировать массив imgpixels по той причине, что следующий пиксель в строке развертки будет пытаться использовать исходное значение для предыдущего пикселя, которое уже может быть отфильтровано.

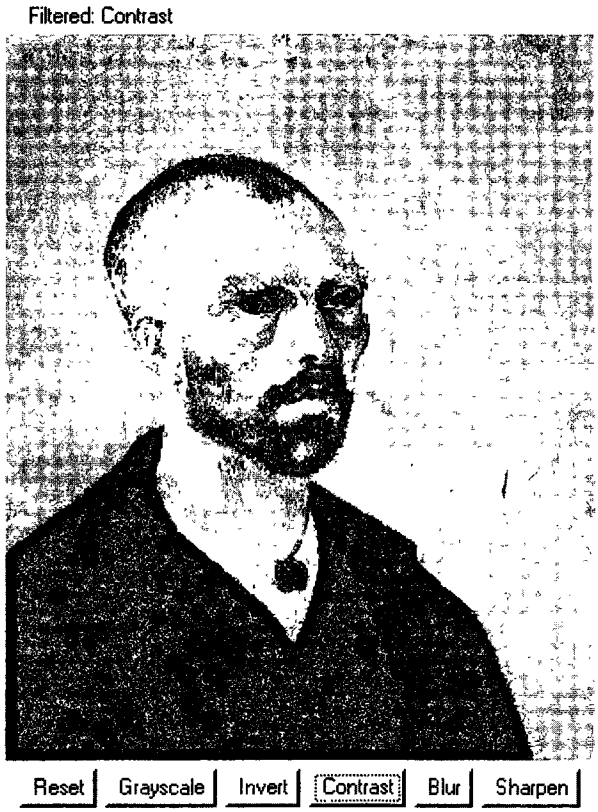


Рис. 26.9. Использование фильтра класса `Contrast` в аплете `ImageFilterDemo`

Два конкретных подкласса, представленных ниже, очень просто реализуют метод `convolver()`, используя массив `imgpixels` для хранения исходных данных и массив `newimgpixels` — для результатов.

```
import java.applet.*;
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 начинается фильтр...

    public Image filter(Applet a, Image in) {
        imageReady = false;
        in.getSource().startProduction(this);

        waitForImage();
        newimgpixels = new int[width*height];

        try {
            convolve();
```

```
        } catch (Exception e) {
            System.out.println("Convolver failed: " + e);
            e.printStackTrace();
        }
        return a.createImage(
            new MemoryImageSource(width, height, newimgpixels,
                                  0, width));
    }

    synchronized void waitForImage() {
        try {
            while(!imageReady) wait();
        } catch (Exception e) {
            System.out.println("Interrupted");
        }
    }

    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;
        }
    }
}
```

Класс `Blur`

Фильтр класса `Blur` является подклассом класса `Convolver` и работает с каждым пикселем в массиве исходного изображения `imgpixels`, вычисляя среднее по окружающей его области размером 3×3. Соответствующий выходной пиксель в массиве `newimgpixels` является подсчитанным средним значением.

```
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. 26.10 показан аплет после работы фильтра `Blur`.

Класс `Sharpen`

Фильтр класса `Sharpen` тоже является подклассом класса `Convolver` и в той или иной степени прямой противоположностью классу `Blur`. Он работает с каждым пикселем в массиве исходного изображения `imgpixels` и вычисляет среднее по окружающей его области размером 3×3. Соответствующий выходной пиксель в массиве `newimgpixels` отличается от центрального пикселя и окружающего среднего, добавленного в него. По сути, это свидетельствует о том, что если пиксель ярче на 30, чем его окружающие пиксели, то остальные пиксели станут ярче на 30. Если же он темнее на 10, то остальные станут темнее на 10. В результате края будут акцентированы, а внутренние участки останутся неизмененными.

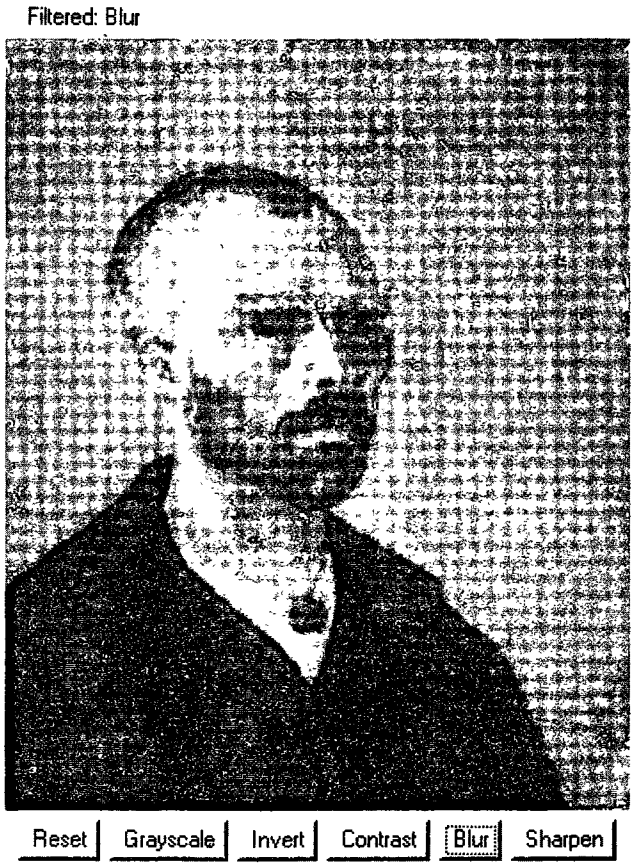


Рис. 26.10. Использование фильтра класса `Blur`в аплете `ImageFilterDemo`

```
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. 26.11 показан аплет после работы фильтра класса `Sharpen`.

Рис. 26.11. Использование фильтра класса `Sharpen` в аплете `ImageFilterDemo`

Аппликационная анимация

Теперь, когда вы уже знаете об особенностях интерфейсов API для работы с изображениями, можем построить интересный аплет, который будет отображать последовательность анимационных ячеек. Анимационные ячейки берутся из одного изображения, которое может компоновать ячейки в сетке, определяемой параметрами `rows` и `cols` дескриптора `<param>`. Каждая ячейка в изображении разделяется подобно тому, как это было использовано ранее в примере аплета `TileImage`. Мы получаем последовательность, в которой будут отображаться ячейки, из параметра `sequence` дескриптора `<param>`. Она представляет собой список номеров ячеек, разделенных запятыми, которые начинаются с нуля и следуют далее по сетке слева направо и сверху вниз.

После того как аплет проанализирует дескрипторы `<param>` и загрузит исходное изображение, он расчленит изображение на несколько небольших второстепенных изображений. Затем запускается поток, благодаря которому изображения будут отображаться в порядке, описанном в параметре `sequence`. Поток бездействует достаточно времени, чтобы поддерживать частоту кадров `framerate`. Ниже показан исходный код.

```
// Пример анимации.
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.util.*;

public class Animation extends Applet implements Runnable {
    Image cell[];
    final int MAXSEQ = 64;
    int sequence[];
    int nseq;
    int idx;
    int framerate;
    volatile boolean stopFlag;

    private int intDef(String s, int def) {
        int n = def;
        if (s != null)
            try {
                n = Integer.parseInt(s);
            } catch (NumberFormatException e) {
                System.out.println("Number Format Exception");
            }
            return n;
        }

        public void init() {
            framerate = intDef(getParameter("framerate"), 5);
            int tilex = intDef(getParameter("cols"), 1);
            int tiley = intDef(getParameter("rows"), 1);
            cell = new Image[tilex*tiley];

            StringTokenizer st =
                   new StringTokenizer(getParameter("sequence"), ",");
            sequence = new int[MAXSEQ]
            nseq = 0;
            while(st.hasMoreTokens() && nseq < MAXSEQ) {
                sequence[nseq] = intDef(st.nextToken(), 0);
                nseq++;
            }
```

```
        try {
            Image img = getImage(getDocumentBase(),
                                 getParameter("img"));
            MediaTracker t = new MediaTracker(this);
            t.addImage(img, 0);
            t.waitForID(0);
            int iw = img.getWidth(null);
            int ih = img.getHeight(null);
            int tw = iw / tilex;
            int th = ih / tiley;
            CropImageFilter f;
            FilteredImageSource fis;

            for (int y=0; y<tiley; y++) {
                for (int x=0; x<tilex; x++) {
                    f = new CropImageFilter(tw*x, th*y, tw, th);
                    fis = new FilteredImageSource(img.getSource(),
                                                  f);
                    int i = y*tilex+x;
                    cell[i] = createImage(fis);
                    t.addImage(cell[i], i);
                }
            }
            t.waitForAll();
        } catch (InterruptedException e) {
            System.out.println("Image Load Interrupted");
        }
    }

    public void update(Graphics g) { }

    public void paint(Graphics g) {
        g.drawImage(cell[sequence[idx]], 0, 0, null);
    }

    Thread t;
    public void start() {
        t = new Thread(this);
        stopFlag = false;
        t.start();
    }

    public void stop() {
        stopFlag = true;
    }

    public void run() {
        idx = 0;
        while (true) {
            paint(getGraphics());
            idx = (idx + 1) % nseq;
            try {
                Thread.sleep(1000/framerate);
            } catch (InterruptedException e) {
                System.out.println("Animation Interrupted");
                return;
            }
            if(stopFlag)
                return;
        }
    }
```

Следующий дескриптор аплета иллюстрирует известное учение Эдварда Майбриджа (Eadweard Muybridge) о движении, которое гласит, что в действительности лошади касаются земли всеми четырьмя копытами. (Естественно, в вашем аплете вы можете использовать другой файл с изображением.)

```
<applet code=Animation width=67 height=48>
<param name=img value=horse.gif>
<param name=rows value=3>
<param name=cols value=4>
<param name=sequence value=0,1,2,3,4,5,6,7,8,9,10>
<param name=framerate value=15>
</applet>
```

На рис. 26.12 показано выполнение аплета. Обратите внимание на то, что в исходном изображении, загруженном после аплета, используется обычный дескриптор `<img>`.

Рис. 26.12. Результат выполнения аплета `Animation`

Дополнительные классы обработки изображений

Кроме классов обработки изображений, описанных в этой главе, пакет `java.awt.image` содержит несколько других классов, которые предлагают расширенные возможности управления процессом визуализации изображений и поддерживают усовершенствованные технологии визуализации. Имеется также пакет визуализации изображений `javax.imageio`, поддерживающий дополнения, с помощью которых можно реализовать обработку различных форматов изображений. Если вас интересуют возможности усовершенствованного графического вывода, придется изучить дополнительные классы, которые можно найти в пакетах `javax.awt.image` и `javax.imageio`.

ГЛАВА 27

Параллельные утилиты

В языке Java с самого начала была предусмотрена встроенная поддержка многопоточности и синхронизации. Например, новые потоки можно создавать за счет реализации интерфейса `Runnable` или расширения класса `Thread`, синхронизация доступна при использовании ключевого слова `synchronized`, а межпотоковые коммуникации поддерживаются методами `wait()` и `notify()`, которые определены в классе `Object`. Вообще говоря, эта встроенная поддержка многопоточности была одним из наиболее важных новшеств в Java и остается одной из главных ее сильных сторон.

Однако какой бы концептуально "чистой" ни была поддержка многопоточности в Java, она не является идеальной для всех приложений, особенно для тех, в которых широко используется множество потоков. Например, первоначальная поддержка многопоточности лишена некоторых высокоуровневых функциональных возможностей (например, семафоров, пулов потоков и диспетчеров), которые способствуют созданию программ, работающих в параллельном режиме.

Необходимо понять с самого начала, что многопоточность используется во многих программах Java, которые в результате становятся "параллельными" (concurrent). Например, многопоточность используется во многих аплетах и сервлетах. Однако что касается настоящей главы, то термин *параллельная программа* относится к программе, которая в *полной мере* использует параллельно выполняющиеся потоки, являющиеся ее *неотъемлемой частью*. Примером является программа, где отдельные потоки служат для одновременного вычисления частичных результатов, используемых в более крупных расчетах. Другим примером является программа, координирующая активность нескольких потоков, каждый из которых пытается обратиться к информации в базе данных. В этом случае доступ в режиме только для чтения может обрабатываться отдельно от доступа, при котором необходимо обеспечить чтение и запись.

Вначале для поддержки параллельных программ в комплекте JDK 5 были добавлены *параллельные утилиты*, которые также часто упоминаются как *параллельный API*. Исходный набор параллельных утилит предоставлял множество возможностей, о которых уже давно мечтали программисты, занимающиеся разработкой параллельных приложений. Например, они предлагают такие средства синхронизации, как семафоры, циклические барьеры, защелки с обратным отсчетом, пулы потоков, диспетчеры выполнения, блокировки, множество параллельных коллекций, а также элегантный способ использования потоков для получения результатов вычислений.

Хотя первоначально параллельные API были внушительны сами по себе, они были существенно расширены в комплекте JDK 7. Самое важное добавление — это инфраструктура *Fork/Join Framework*, которая облегчает создание программ, использующих несколько процессоров (таких, как в многоядерных системах). Таким

образом, она упрощает разработку программ, в которых две или больше частей на самом деле выполняются одновременно (т.е. имеет место истинное параллельное выполнение), а не за счет квантования времени. Как вы легко можете представить себе, параллельное выполнение может существенно увеличить скорость определенных операций. Поскольку многоядерные системы распространены уже широко, включение инфраструктуры Fork/Join Framework столь же своевременно, сколь и мощно.

Первоначальные параллельные API были весьма обширны, а инфраструктура Fork/Join Framework еще увеличивает их размер. Как и следовало ожидать, большинство проблем, связанных с использованием параллельности, весьма сложны. Обсуждение всех их аспектов выходит за рамки этой книги. Несмотря на это, всем программистам имеет смысл иметь общее представление и практические знания о параллельных API. Даже в тех программах, которые не используют параллельность интенсивно, такие средства, как синхронизаторы, вызываемые потоки и исполнители, применимы к различным ситуациям. Возможно, важнее всего то, что в связи с распространением многоядерных компьютеров решения, задействующие инфраструктуру Fork/Join Framework, также станут весьма распространены. Поэтому данная глава представляет краткий обзор утилит параллельности и демонстрирует несколько примеров их использования. Она завершается углубленным исследованием инфраструктуры Fork/Join Framework.

Пакеты параллельного API

Параллельные утилиты содержатся в пакете `java.util.concurrent` и двух его вложенных пакетах, `java.util.concurrent.atomic` и `java.util.concurrent.locks`. Далее следует краткий обзор их содержимого.

Пакет `java.util.concurrent`

Пакет `java.util.concurrent` определяет основные функциональные возможности, которые поддерживают альтернативные варианты встроенных методов синхронизации и межпотоковых коммуникаций. Он определяет следующие ключевые элементы:

- синхронизаторы;
- исполнители;
- параллельные коллекции;
- инфраструктуру Fork/Join Framework.

Синхронизаторы предлагают высокоуровневые способы синхронизации взаимодействий между несколькими потоками. В пакете `java.util.concurrent` определен набор классов *синхронизаторов*, описанных в табл. 27.1.

Обратите внимание на то, что каждый синхронизатор предлагает решение задачи синхронизации определенного рода. Благодаря этому можно оптимизировать работу каждого синхронизатора. Раньше эти типы объектов синхронизации необходимо было создавать вручную. Параллельный API стандартизирует их и делает доступными для всех программистов, работающих с Java.

Таблица 27.1. Классы синхронизаторов, определенные в пакете `java.util.concurrent`

Класс	Описание
`Semaphore`	Реализует классический семафор
`CountDownLatch`	Ожидание длится до тех пор, пока не произойдет определенное количество событий
`CyclicBarrier`	Позволяет группе потоков войти в режим ожидания в предварительно заданной точке выполнения
`Exchanger`	Осуществляет обмен данными между двумя потоками
`Phaser`	Синхронизирует потоки, проходящие через несколько фаз операций. (Добавлено в JDK 7)

Исполнители (executor) управляют выполнением потоков. Первым в иерархии исполнителей является интерфейс `Executor`, который служит для запуска потока. Интерфейс `ExecutorService` расширяет интерфейс `Executor` и предлагает методы, управляющие исполнением. Существует три реализации интерфейса `ExecutorService`: классы `ThreadPoolExecutor`, `ScheduledThreadPoolExecutor` и класс `ForkJoinPool`, добавленный в JDK 7. Пакет `java.util.concurrent` также определяет служебный класс `Executors`, который содержит несколько статических методов, упрощающих создание разнообразных исполнителей.

С исполнителями связаны интерфейсы `Future` и `Callable`. Интерфейс `Future` содержит значение, возвращаемое потоком после его выполнения. Таким образом, его значение определяется "в будущем", когда поток завершит свое выполнение. Интерфейс `Callable` определяет поток, возвращающий значение.

Пакет `java.util.concurrent` определяет несколько классов параллельных коллекций, включая `ConcurrentHashMap`, `ConcurrentLinkedQueue` и `CopyOnWriteArrayList`. Они предлагают параллельные альтернативные варианты для связанных с ними классов, определенных в инфраструктуре Collections Framework.

Инфраструктура Fork/Join Framework поддерживает параллельное программирование. Ее основные классы – `ForkJoinTask`, `ForkJoinPool`, `RecursiveTask` и `RecursiveAction`. Как упоминалось, инфраструктура Fork/Join Framework была добавлена комплектом JDK 7.

Наконец, для улучшенной обработки синхронизации потоков пакет `java.util.concurrent` определяет перечисление `TimeUnit`.

Пакет `java.util.concurrent.atomic`

Этот пакет упрощает использование переменных в параллельной среде. Он предлагает средства эффективного обновления значений переменной без применения блокировок. Для этого используются такие классы, как `AtomicInteger` и `AtomicLong`, а также методы вроде `compareAndSet()`, `decrementAndGet()` и `getAndSet()`. Эти методы работают в режиме одной непрерывной операции.

Пакет `java.util.concurrent.locks`

Пакет `java.util.concurrent.locks` предлагает альтернативный вариант работы с синхронизированными методами. В его основе лежит интерфейс `Lock`,

определяющий основной механизм, который используется для получения доступа к объекту и отказа в доступе. Ключевыми методами являются `lock()`, `tryLock()` и `unlock()`. Преимущество этих методов состоит в том, что они расширяют возможности управления синхронизацией.

Далее в главе займемся подробным рассмотрением компонентов параллельного API.

Использование объектов синхронизации

Объекты синхронизации представлены классами `Semaphore`, `CountDownLatch`, `CyclicBarrier`, `Exchanger` и `Phaser`. Вместе они позволяют без особых сложностей решать некоторые задачи синхронизации, справиться с которыми ранее было довольно непросто. Их также можно применять к широкому диапазону программ — даже к тем, которые поддерживают только ограниченный параллелизм. Поскольку объекты синхронизации могут встречаться практически во всех программах Java, остановимся на них более подробно.

Класс `Semaphore`

Первым объектом синхронизации, который могут сразу же вспомнить большинство читателей, является класс `Semaphore`, реализующий классический семафор. *Семафор* управляет доступом к общему ресурсу с помощью счетчика. Если счетчик больше нуля, доступ разрешается. Если он равен нулю, в доступе будет отказано. В действительности этот счетчик подсчитывает *разрешения*, открывающие доступ к общему ресурсу. Следовательно, чтобы получить доступ к ресурсу, поток должен получить разрешение на доступ у семафора.

В общем случае, чтобы использовать семафор, поток, которому требуется доступ к общему ресурсу, пытается получить разрешение. Если значение счетчика семафора будет больше нуля, поток получит разрешение, после чего значение счетчика семафора уменьшается на единицу. В противном случае поток будет заблокирован до тех пор, пока не сможет получить разрешение. Если доступ к общему ресурсу потоку больше не нужен, он освобождает разрешение, в результате чего значение счетчика семафора увеличивается на единицу. Если в это время другой поток ожидает разрешения, то он сразу же его получает. Описанный механизм реализуется в Java классом `Semaphore`.

Класс `Semaphore` имеет два конструктора, показанных далее.

```
Semaphore(int число)
Semaphore(int число, boolean как)
```

Здесь параметр *число* указывает исходное значение счетчика разрешений. Таким образом, *число* определяет количество потоков, которым может быть предоставлен доступ к общему ресурсу одновременно. Если параметр *число* содержит значение `1`, только один поток сможет обратиться к ресурсу. По умолчанию ожидающим потокам предоставляется разрешение в неопределенном порядке. Если параметру *как* присвоить значение `true`, то вы тем самым определите, что ожидающим потокам будут выдаваться разрешения в том порядке, в каком они запрашивали доступ.

Чтобы получить разрешение, вызовите метод `acquire()`, который имеет следующие две формы.

```
void acquire() throws InterruptedException
void acquire(int число) throws InterruptedException
```

Первая форма запрашивает одно разрешение, а вторая — *число* разрешений. Обычно используется первая форма. Если разрешение не будет предоставлено во время вызова метода, то вызывающий поток будет приостановлен до тех пор, пока не будет получено разрешение.

Чтобы освободить разрешение, вызовите метод `release()`, который имеет следующие две формы.

```
void release()
void release(int число)
```

Первая форма освобождает одно разрешение, а вторая — то количество разрешений, которое указано в параметре *число*.

Чтобы использовать семафор для управления доступом к ресурсу, каждый поток, которому необходимо использовать этот ресурс, должен вначале вызвать метод `acquire()`, прежде чем обращаться к ресурсу. Когда поток завершает свою работу с ресурсом, он должен вызвать метод `release()`. Ниже приведен пример использования семафора.

```
// Пример простого семафора.

import java.util.concurrent.*;

class SemDemo {

    public static void main(String args[]) {
        Semaphore sem = new Semaphore(1);

        new IncThread(sem, "A");
        new DecThread(sem, "B");
    }
}

// Общий ресурс.
class Shared {
    static int count = 0;
}

// Поток выполнения, увеличивающий значение счетчика на единицу.
class IncThread implements Runnable {
    String name;
    Semaphore sem;
    IncThread(Semaphore s, String n) {
        sem = s;
        name = n;
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Запуск " + name);
        try {

            // Сначала получаем разрешение.
            System.out.println(name + " ожидает разрешения.");
            sem.acquire();
            System.out.println(name + " получает разрешение.");
            // Теперь обращаемся к общему ресурсу.

            for(int i=0; i < 5; i++) {
                Shared.count++;
                System.out.println(name + ": " + Shared.count);
```

```
                // Если это возможно, разрешаем контекстное переключение.
                Thread.sleep(10);
            }
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }

        // Освобождаем разрешение.
        System.out.println(name + " освобождает разрешение.");
        sem.release();
    }
}

// Поток выполнения, уменьшающий значение счетчика на единицу.
class DecThread implements Runnable {
    String name;
    Semaphore sem;

    DecThread(Semaphore s, String n) {
        sem = s;
        name = n;
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Starting " + name);

        try {
            // Сначала получаем разрешение.
            System.out.println(name + " ожидает разрешения.");
            sem.acquire();
            System.out.println(name + " получает разрешение.");

            // Теперь обращаемся к общему ресурсу.
            for(int i=0; i < 5; i++) {
                Shared.count--;
                System.out.println(name + ": " + Shared.count);

                // Если это возможно, разрешаем контекстное переключение.
                Thread.sleep(10);
            }
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }

        // Освобождаем разрешение.
        System.out.println(name + " освобождает разрешение.");
        sem.release();
    }
}
```

Ниже показаны результаты выполнения этой программы. (Конкретный порядок выполнения потоков может быть иным.)

```
Запуск A
A ожидает разрешения.
A получает разрешение.
A: 1
Запуск B
B ожидает разрешения.
```

```
A: 2
A: 3
A: 4
A: 5
A освобождает разрешение.
B получает разрешение.
B: 4
B: 3
B: 2
B: 1
B: 0
B освобождает разрешение.
```

В программе используется семафор для управления доступом к переменной count, которая является статической переменной класса Shared. Значение переменной Shared.count увеличивается на 5 в методе run() класса IncThread и уменьшается на 5 в одноименном методе класса DecThread. Для защиты этих двух потоков от одновременного доступа к переменной Shared.count, доступ предоставляется только после того, как будет получено разрешение от управляющего семафора. После того как доступ будет завершен, разрешение освобождается. Таким образом, только один поток в одно и то же время получит доступ к переменной Shared.count, что можно видеть из результатов вывода.

Обратите внимание на то, что в методе run() классов IncThread и DecThread вызывается метод sleep(). Он "гарантирует", что доступ к переменной Shared.count будет синхронизироваться семафором. В методе run() вызов метода sleep() приводит к тому, что вызывающий поток будет приостанавливаться в промежутках между каждым доступом к переменной Shared.count. Как правило, благодаря этому должен выполняться второй поток. Однако поскольку мы работаем с семафором, то второй поток должен ожидать до тех пор, пока первый поток не освободит разрешение, а это происходит только после того, как будут завершены все доступы, производимые первым потоком. Таким образом, значение переменной Shared.count вначале увеличивается на 5 в объекте класса IncThread, а затем уменьшается на 5 в объекте класса DecThread. Увеличения и уменьшения значений происходят *строго по порядку*.

Если бы мы не использовали семафор, то доступы к переменной Shared.count, производимые каждым потоком, осуществлялись бы одновременно, поэтому увеличение и уменьшение значения происходило бы вперемешку. Чтобы убедиться в этом, попробуйте закомментировать вызовы методов acquire() и release(). Запустив программу, вы увидите, что доступ к переменной Shared.count больше не является синхронизированным и каждый поток обращается к переменной Shared.count сразу же, как только им выделяется временной интервал.

Несмотря на то что во многих случаях применение семафора не представляет сложности, как это можно было видеть в предыдущей программе, возможны также и более сложные варианты его использования. Ниже показан один из таких примеров. Это переработанная версия программы поставщика и потребителя, представленной в главе 11. Здесь используется два семафора, которые регулируют потоки поставщика и потребителя и гарантируют, что за каждым вызовом метода put() будет следовать соответствующий вызов метода get().

```
// Реализация поставщика и потребителя, использующая
// семафоры для управления синхронизацией.
import java.util.concurrent.Semaphore;

class Q {
    int n;
```

```
    // Начинаем с недоступного семафора потребителя.
    static Semaphore semCon = new Semaphore(0);
    static Semaphore semProd = new Semaphore(1);

    void get() {
        try {
            semCon.acquire();
        } catch(InterruptedException e) {
            System.out.println("Произошло InterruptedException");
        }

        System.out.println("Получено: " + n);
        semProd.release();
    }

    void put(int n) {
        try {
            semProd.acquire();
        } catch(InterruptedException e) {
            System.out.println("Произошло InterruptedException");
        }
        this.n = n;
        System.out.println("Отправлено: " + n);
        semCon.release();
    }
}

class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    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;
        new Thread(this, "Consumer").start();
    }

    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 Consumer(q);
        new Producer(q);
    }
}
```

Ниже показана часть результатов.

```
Отправлено: 0
Получено: 0
Отправлено: 1
Получено: 1
Отправлено: 2
Получено: 2
Отправлено: 3
Получено: 3
Отправлено: 4
Получено: 4
Отправлено: 5
Получено: 5
.
.
.
```

Как видите, здесь происходит синхронизация вызовов методов `put()` и `get()`. То есть после каждого вызова метода `put()` следует вызов метода `get()`, поэтому ни одно значение не может быть пропущено. Если бы семафоры не использовались, вызовы метода `put()` могли бы происходить без чередования с вызовами метода `get()`, в результате чего значения были бы пропущены. (Чтобы убедиться в этом, удалите код семафора и посмотрите на полученные результаты.)

Последовательность вызовов методов `put()` и `get()` обрабатывается двумя семафорами: `semProd` и `semCon`. Прежде чем метод `put()` сможет произвести значение, он должен получить разрешение от семафора `semProd`. После того как значение будет определено, он освобождает семафор `semProd`. Прежде чем метод `get()` сможет использовать значение, он должен получить разрешение от семафора `semCon`. После того как он закончит работу с этим значением, он освобождает семафор `semCon`. Такой механизм "передачи и получения" гарантирует, что за каждым вызовом метода `put()` будет следовать вызов метода `get()`.

Обратите внимание на то, что семафор `semCon` инициализируется без доступных разрешений. Поэтому метод `put()` выполняется первым. Возможность задавать исходное состояние синхронизации является одной из наиболее ярких характеристик семафоров.

Класс `CountDownLatch`

Иногда необходимо, чтобы поток находился в режиме ожидания до тех пор, пока не произойдет одно или несколько событий. Для этих целей параллельный API предлагает защелку класса `CountDownLatch`. Этот класс изначально создается с количеством событий, которые должны произойти до того момента, как будет снята защелка. Каждый раз, когда происходит событие, значение счетчика уменьшается. Когда значение счетчика станет равным нулю, защелка будет снята.

Класс `CountDownLatch` имеет следующий конструктор.

```
CountDownLatch(int число)
```

Здесь *число* определяет количество событий, которые должны произойти до того, как будет снята защелка.

Для обслуживания защелки поток вызывает метод `await()`, который имеет следующие формы.

```
void await() throws InterruptedException
boolean await(long ожидать, TimeUnit ту) throws InterruptedException
```

В первом случае ожидание длится до тех пор, пока значение счета, связанного с вызывающим объектом класса `CountDownLatch`, не станет равно нулю. Во втором случае ожидание длится только в течение определенного периода време-

ни, определенного параметром *ожидать*. Единицы, представляемые этим параметром, определяются параметром *tu*, который является объектом перечисления TimeUnit. (Перечисление TimeUnit рассматривается далее в этой главе.) Метод возвращает значение false, если достигнут срок, и значение true — если обратный отсчет достигает нуля.

Чтобы известить о событии, нужно вызвать метод countDown().

```
void countDown()
```

При каждом вызове метода countDown() значение счета, связанного с вызывающим объектом, уменьшается на единицу.

В следующей программе представлен пример применения класса CountDownLatch. В ней создается защелка, снять которую можно будет только по прошествии пяти событий.

```
// Демонстрация применения CountDownLatch.

import java.util.concurrent.CountDownLatch;

class CDLDemo {
    public static void main(String args[]) {
        CountDownLatch cdl = new CountDownLatch(5);

        System.out.println("Запуск");

        new MyThread(cdl);

        try {
            cdl.await();
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }
        System.out.println("Завершение");
    }
}

class MyThread implements Runnable {
    CountDownLatch latch;

    MyThread(CountDownLatch c) {
        latch = c;
        new Thread(this).start();
    }

    public void run() {
        for(int i = 0; i<5; i++) {
            System.out.println(i);
            latch.countDown(); // обратный отсчет
        }
    }
}
```

Ниже показаны результаты выполнения этой программы.

```
Запуск
0
1
2
3
4
Завершение
```

Внутри метода `main()` создается защелка класса `CountDownLatch` по имени `cdl`, исходное значение которой равно 5. После этого создается экземпляр класса `MyThread`, который начинает выполнение нового потока. Обратите внимание на то, что защелка `cdl` передается в качестве параметра конструктору класса `MyThread` и сохраняется в переменной экземпляра `latch`. Затем главный поток вызывает метод `await()` для защелки `cdl`, в результате чего выполнение главного потока приостанавливается до тех пор, пока счетчик защелки `cdl` пять раз не уменьшится на единицу.

Внутри метода `run()` конструктора класса `MyThread` создается цикл, который повторяется пять раз. Во время каждого повторения вызывается метод `countDown()` для переменной экземпляра `latch`, который ссылается на защелку `cdl` в методе `main()`. После пятого повторения защелка снимается, позволяя возобновить главный поток.

Класс `CountDownLatch` является мощным и простым в использовании объектом синхронизации, который будет полезен в тех случаях, когда поток должен находиться в режиме ожидания, пока не произойдет одно или несколько событий.

Класс `CyclicBarrier`

В программировании нередко возникают такие ситуации, когда два или более потоков должны находиться в режиме ожидания в предварительно определенной точке выполнения до тех пор, пока все потоки из этого набора не достигнут этой точки. Для этого параллельные API предлагают класс `CyclicBarrier`. Он позволяет определить объект синхронизации, который приостанавливается до тех пор, пока определенное количество потоков не достигнет барьерной точки.

Класс `CyclicBarrier` имеет следующие два конструктора.

```
CyclicBarrier(int количПотоков)
CyclicBarrier(int количПотоков, Runnable действие)
```

Здесь параметр *количПотоков* определяет количество потоков, которые должны достигнуть барьера до того, как выполнение будет продолжено. Во втором варианте параметр *действие* определяет поток, который будет выполняться по достижении барьера.

Общая процедура использования класса `CyclicBarrier` выглядит следующим образом. В первую очередь необходимо создать объект класса `CyclicBarrier`, указав количество потоков для ожидания. Затем, когда каждый поток достигнет барьера, нужно вызвать метод `await()` для данного объекта. В результате этого выполнение потока будет приостановлено до тех пор, пока все остальные потоки также не вызовут метод `await()`. После того как указанное количество потоков достигнет барьера, метод `await()` вернет результат и выполнение будет возобновлено. Кроме того, если определить какое-нибудь действие, то этот поток будет выполнен.

Метод `await()` имеет следующие две формы.

```
int await() throws InterruptedException, BrokenBarrierException
int await(long ожидать, TimeUnit tu) throws InterruptedException,
BrokenBarrierException, TimeoutException
```

В первом случае ожидание длится до тех пор, пока каждый поток не достигнет барьерной точки. Во втором случае ожидание длится в течение определенного периода времени *ожидать*. Единицы времени этого параметра определяются параметром *tu*. В обоих случаях возвращается значение, показывающее порядок, в соответствии с которым потоки будут достигать барьерной точки. Первый поток возвращает значение, равное количеству ожидаемых потоков минус 1. Последний поток возвращает нуль.

Ниже показан пример, иллюстрирующий класс CyclicBarrier. Он ожидает, пока совокупность трех потоков не достигнет барьерной точки. После того как это произойдет, будет выполнен поток, определяемый при помощи класса BarAction.

```
// Демонстрация применения CyclicBarrier.

import java.util.concurrent.*;

class BarDemo {
    public static void main(String args[]) {
        CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );

        System.out.println("Запуск");

        new MyThread(cb, "A");
        new MyThread(cb, "B");
        new MyThread(cb, "C");
    }
}

// Поток выполнения, использующий CyclicBarrier.
    .
class MyThread implements Runnable {
    CyclicBarrier cbar;
    String name;

    MyThread(CyclicBarrier c, String n) {
        cbar = c;

        name = n;
        new Thread(this).start();
    }

    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 после завершения выполнения
CyclicBarrier.
class BarAction implements Runnable {
    public void run() {
        System.out.println("Барьер достигнут!");
    }
}
```

Ниже показаны результаты выполнения.

```
Запуск
A
B
C
```

```
Барьер достигнут!
```

Класс `CyclicBarrier` можно использовать повторно, так как он освобождает ожидающие потоки каждый раз, когда определенное количество потоков вызывает метод `await()`. Например, если метод `main()` в предыдущей программе изменить следующим образом:

```
public static void main(String args[]) {
    CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );

    System.out.println("Запуск");

    new MyThread(cb, "A");
    new MyThread(cb, "B");
    new MyThread(cb, "C");
    new MyThread(cb, "X");
    new MyThread(cb, "Y");
    new MyThread(cb, "Z");
}
```

то результат выполнения будет таким.

```
Запуск
A
B
C
Барьер достигнут!
X
Y
Z
Барьер достигнут!
```

Как можно видеть из предыдущего примера, класс `CyclicBarrier` предлагает элегантное решение ранее сложной задачи.

Класс `Exchanger`

Вероятно, наиболее интересным классом синхронизации является класс `Exchanger`, предназначенный для упрощения процесса обмена данными между двумя потоками. Работа класса `Exchanger` довольно проста: он просто ожидает, пока два отдельных потока не вызовут его метод `exchange()`. Как только это произойдет, он произведет обмен данными, имеющимися у обоих потоков. В своем использовании этот механизм является одновременно и элегантным, и простым. Варианты применения класса `Exchanger` можно представить очень просто. Например, один поток подготавливает буфер для получения информации через сетевое соединение. Другой поток заполняет этот буфер информацией, получаемой при помощи соединения. Оба потока работают совместно, поэтому каждый раз, когда возникает необходимость в использовании нового буфера, осуществляется обмен данными.

Класс `Exchanger` является обобщенным классом и имеет следующее объявление.

```
Exchanger<V>
```

Здесь параметр *V* определяет тип данных для обмена.

Класс `Exchanger` определяет единственный метод `exchange()`, который имеет следующие две формы.

```
V exchange(V буфер) throws InterruptedException
V exchange(V буфер, long ожидать, TimeUnit ту) throws
InterruptedException, TimeoutException
```

Здесь параметр *буфер* представляет ссылку на данные для обмена. Данные, полученные из другого потока, возвращаются. Вторая форма метода `exchange()`

позволяет определить время простоя. Ключевая особенность метода `exchange()` заключается в том, что его выполнение не будет продолжено до тех пор, пока он не будет вызван для одного и того же объекта класса `Exchanger` двумя отдельными потоками. Таким образом, метод `exchange()` синхронизирует обмен данными.

Ниже показан пример применения класса `Exchanger`. В нем создается два потока. Один поток создает пустой буфер, который получает данные, занесенные в него другим потоком. Таким образом, первый поток меняет пустой поток на полный.

```
// Пример использования класса Exchanger.

import java.util.concurrent.Exchanger;

class ExgrDemo {
    public static void main(String args[]) {
        Exchanger<String> exgr = new Exchanger<String>();
        new UseString(exgr);
        new MakeString(exgr);
    }
}

// Поток Thread, формирующий строку.
class MakeString implements Runnable {
    Exchanger<String> ex;
    String str;
    MakeString(Exchanger<String> c) {
        ex = c;
        str = new String();
        new Thread(this).start();
    }

    public void run() {
        char ch = 'A';
        for(int i = 0; i < 3; i++) {
            // Заполнение буфера
            for(int j = 0; j < 5; j++)
                str += (char) ch++;
            try {
                // Заполненный буфер становится пустым.
                str = ex.exchange(str);
            } catch(InterruptedException exc) {
                System.out.println(exc);
            }
        }
    }
}

// Поток Thread, использующий строку.
class UseString implements Runnable {
    Exchanger<String> ex;
    String str;
    UseString(Exchanger<String> c) {
        ex = c;
        new Thread(this).start();
    }

    public void run() {

        for(int i=0; i < 3; i++) {
            try {
                // Пустой буфер становится заполненным.
```

```
                str = ex.exchange(new String());
                System.out.println("Получено: " + str);
            } catch(InterruptedException exc) {
                System.out.println(exc);
            }
        }
    }
}
```

Ниже показаны результаты выполнения этой программы.

```
Получено: ABCDE
Получено: FGHIJ
Получено: KLMNO
```

В этой программе метод `main()` создает объект класса `Exchanger` для строк. Этот объект затем служит для синхронизации обмена строками между классами `MakeString` и `UseString`. Класс `MakeString` заполняет строку данными. Класс `UseString` заполняет пустой буфер, а затем отображает содержимое только что созданной строки. Замена пустого буфера полным синхронизируется с помощью метода `exchange()`, который вызывается методом `run()` обоих классов.

Класс `Phaser`

Комплект JDK 7 содержит новый класс синхронизации `Phaser`. Его главная задача — обеспечение синхронизации потоков, которые представляют одну или несколько фаз действия. Например, у вас может быть ряд потоков, которые реализуют три фазы приложения обработки заказов. На первой фазе используются отдельные потоки, чтобы проверить информацию клиента, товар и его цену. По завершении этой фазы, у второй фазы есть два потока, которые вычисляют стоимость доставки и сумму соответствующего налога. Затем, на заключительной фазе, подтверждается оплата и определяется ориентировочное время доставки. В прошлом, чтобы синхронизировать несколько потоков, вовлеченных в этот сценарий, потребовалось бы выполнение некоторой работы с вашей стороны. С появлением класса `Phaser` процесс существенно упростился.

Для начала имеет смысл узнать, что класс `Phaser` работает подобно описанному ранее классу `CyclicBarrier`, за исключением того, что он поддерживает несколько фаз. В результате класс `Phaser` позволяет определить объект синхронизации, который ожидает завершения определенной фазы. Затем он переходит к следующей фазе и снова ожидает ее завершения. Следует понять, что класс `Phaser` может также использоваться для синхронизации только одной фазы. В этом отношении он действует подобно классу `CyclicBarrier`. Однако его главная задача — синхронизация нескольких фаз. В классе `Phaser` определено четыре конструктора. В данном разделе используются следующие два.

```
Phaser()
Phaser(int количСторон)
```

Первый конструктор создает *фазер* (phaser) с нулевым регистрационным счетом. Второй устанавливает значение регистрационного счета равным *количСторон*. К регистрируемым фазером объектам зачастую применяется термин *сторона* (party). Хотя обычно есть полное соответствие между количеством регистрантов и количеством синхронизируемых потоков, это не обязательно. В обоих случаях текущая фаза нулевая. Таким образом, когда создается экземпляр класса `Phaser`, он первоначально находится в нулевой фазе.

В общем, класс Phaser используется так. Сначала создается новый экземпляр класса Phaser. Затем на фазере регистрируется одна или несколько сторон вызовом метода register() или при указании необходимого количества сторон в конструкторе. Для каждой зарегистрированной стороны имеется фазер, ожидающий, пока все зарегистрированные стороны не закончат фазу. Сторона сообщает об этом при вызове одного из множества методов, предоставляемых классом Phaser, таких как метод arrive() или arriveAndAwaitAdvance(). Как только все стороны готовы, фаза считается законченной и фазер может перейти к следующей фазе (если она есть) или завершить работу. Следующие разделы объясняют процесс подробнее.

Для регистрации стороны после создания объекта класса Phaser вызовите метод register().

```
int register()
```

В результате он возвратит номер зарегистрированной фазы.

Чтобы сообщить о завершении фазы, сторона должна вызвать метод arrive() или некий его вариант. Когда количество завершений равняется количеству зарегистрированных сторон, фаза заканчивается и объект класса Phaser переходит к следующей фазе (если она есть). Метод arrive() имеет следующую общую форму.

```
int arrive()
```

Этот метод сообщает о том, что сторона (обычно поток выполнения) закончила некую задачу (или часть задачи). Он возвращает текущий номер фазы. Если работа фазера закончена, то он возвращает отрицательное значение. Метод arrive() не приостанавливает выполнение вызывающего потока. Это значит, что он не ожидает завершения фазы. Этот метод должен быть вызван только зарегистрированной стороной.

Если вы хотите указать на завершение фазы, а затем ожидать завершения этой фазы всеми остальными регистрантами, используйте метод arriveAndAwaitAdvance().

```
int arriveAndAwaitAdvance()
```

Он ожидает завершения всех сторон и возвращает номер следующей фазы или отрицательное значение, если фазер завершил работу. Этот метод должен быть вызван только зарегистрированной стороной.

Поток может завершиться, а затем отменить свою регистрацию, вызвав метод arriveAndDeregister().

```
int arriveAndDeregister()
```

Метод возвращает номер текущей фазы или отрицательное значение, если фазер завершил работу. Он не ожидает завершения фазы. Этот метод должен быть вызван только зарегистрированной стороной.

Чтобы получить номер текущей фазы, вызовите метод getPhase(), который выглядит так.

```
final int getPhase()
```

Когда объект класса Phaser будет создан, первая фаза будет иметь номер 0, вторая — 1, третья — 2 и т.д. Если вызывающий объект класса Phaser завершил работу, возвращается отрицательное значение.

Вот пример, демонстрирующий класс Phaser в действии. Здесь создается три потока, каждый из которых имеет три фазы. Для синхронизации каждой фазы используется класс Phaser.

```
// Пример применения класса Phaser.

import java.util.concurrent.*;
```

```
class PhaserDemo {
    public static void main(String args[]) {
        Phaser phsr = new Phaser(1);
        int curPhase;

        System.out.println("Starting");

        new MyThread(phsr, "A");
        new MyThread(phsr, "B");
        new MyThread(phsr, "C");

        // Ожидать завершения всеми потоками фазы один.
        curPhase = phsr.getPhase();
        phsr.arriveAndAwaitAdvance();
        System.out.println("Phase " + curPhase + " Complete");

        // Ожидать завершения всеми потоками фазы два.
        curPhase = phsr.getPhase();
        phsr.arriveAndAwaitAdvance();
        System.out.println("Phase " + curPhase + " Complete");

        curPhase = phsr.getPhase();
        phsr.arriveAndAwaitAdvance();
        System.out.println("Phase " + curPhase + " Complete");

        // Отменить регистрацию основного потока.
        phsr.arriveAndDeregister();

        if(phsr.isTerminated())
            System.out.println("The Phaser is terminated");
    }
}

// Поток выполнения, использующий Phaser.
class MyThread implements Runnable {
    Phaser phsr;
    String name;

    MyThread(Phaser p, String n) {
        phsr = p;
        name = n;
        phsr.register();
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Thread " + name + " Beginning Phase One");
        phsr.arriveAndAwaitAdvance(); // Сигнал завершения.

        // Небольшая пауза для предотвращения перемешанного вывода.
        // Только для иллюстрации. Это не обязательно для правильной
        // работы фазера.
        try {
            Thread.sleep(10);
        } catch(InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("Thread " + name + " Beginning Phase Two");
        phsr.arriveAndAwaitAdvance(); // Сигнал завершения.
```

```
        // Небольшая пауза для предотвращения перемешанного вывода.
        // Только для иллюстрации. Это не обязательно для правильной
        // работы фазера.
        try {
            Thread.sleep(10);
        } catch(InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("Thread " + name + " Beginning Phase Three");
        phsr.arriveAndDeregister(); // Сигнал завершения и
                                    // отмены регистрации.
    }
}
```

Вот вывод.

```
Starting
Thread A Beginning Phase One
Thread C Beginning Phase One
Thread B Beginning Phase One
Phase 0 Complete
Thread B Beginning Phase Two
Thread C Beginning Phase Two
Thread A Beginning Phase Two
Phase 1 Complete
Thread C Beginning Phase Three
Thread B Beginning Phase Three
Thread A Beginning Phase Three
Phase 2 Complete
The Phaser is terminated
```

Давайте внимательнее рассмотрим ключевые разделы программы. Сначала в методе `main()` создается объект `phsr` класса `Phaser` с начальным номером стороны 1 (что соответствует основному потоку). Затем запускается три потока при создании трех объектов класса `MyThread`. Обратите внимание на то, что объекту класса `MyThread` передается ссылка на объект `phsr` (фазер). Объекты класса `MyThread` используют этот фазер для синхронизации своих действий. Затем метод `main()` вызывает метод `getPhase()`, чтобы получить номер текущей фазы (который первоначально является нулевым), а потом метод `arriveAndAwaitAdvance()`. Это приостанавливает метод `main()`, пока не закончится нулевая фаза, а этого не случится, пока все объекты класса `MyThread` также не вызовут метод `arriveAndAwaitAdvance()`. Когда это произойдет, метод `main()` возобновит выполнение, сообщив о завершении нулевой фазы, и перейдет ко второй фазе. Этот процесс повторяется до завершения всех трех фаз. Затем метод `main()` вызывает метод `arriveAndDeregister()`. В этот момент регистрация всех трех объектов класса `MyThread` отменена. Поскольку это приводит к тому, что при переходе фазера к следующей фазе нет никаких зарегистрированных сторон, работа фазера заканчивается.

Теперь рассмотрим объект класса `MyThread`. В первую очередь обратите внимание на то, что конструктору передается ссылка на фазер, который будет использован, а затем регистраторы с новым потоком как сторона этого фазера. Таким образом, каждый новый объект класса `MyThread` становится стороной, зарегистрированной с переданным фазером. Обратите также внимание на то, что у каждого потока есть три фазы. В этом примере каждая фаза состоит из знакоместа, которое просто отображает имя потока и то, что он делает. Безусловно, в реальном коде поток выполнял бы более существенные действия. Между первыми двумя фазами поток вызывает метод `arriveAndAwaitAdvance()`. Таким образом,

каждый поток ожидает завершения фазы всеми потоками (включая основной поток). После завершения всех потоков (включая основной поток) фазер переходит к следующей фазе. После третьей фазы каждый поток отменяет свою регистрацию вызовом метода `arriveAndDeregister()`. Как объясняют комментарии объекта класса `MyThread`, вызов метода `sleep()` используется в иллюстративных целях для предотвращения перемешивания вывода из-за многопоточности. Это не обязательно для правильной работы фазера. Если вы удалите вызовы метода `sleep()`, то вывод может выглядеть немного перепутанным, но фазы все равно будут синхронизированы правильно.

Еще один момент: хотя в приведенном примере использовано три потока одинакового типа, это не обязательное требование. Каждая сторона, которая использует фазер, может быть индивидуальной и выполнять индивидуальные задачи.

Происходящее при переходе к следующей фазе вполне можно взять под свой контроль. Для этого следует переопределить метод `onAdvance()`. Этот метод вызывается средой выполнения, когда фазер переходит от одной фазы к следующей.

```
protected boolean onAdvance(int фаза, int количСторон)
```

Здесь параметр *фаза* будет содержать текущий номер фазы перед его инкрементом, а параметр *количСторон* — количество зарегистрированных сторон. Для завершения работы фазера метод `onAdvance()` должен возвратить значение `true`. Чтобы поддерживать фазер в действии, метод `onAdvance()` должен возвращать значение `false`. Заданная по умолчанию версия метода `onAdvance()` возвращает значение `true` (это завершает работу фазера), когда нет никаких зарегистрированных сторон. Как правило, ваше переопределение также должно следовать этой практике.

Одна из причин переопределения метода `onAdvance()` заключается в том, чтобы позволить фазеру выполнить определенное количество фаз, а затем остановиться. Ниже приведен пример разновидности такого применения. Здесь создается класс по имени `MyPhaser`, который расширяет класс `Phaser` так, чтобы он выполнял определенное количество фаз. Для этого переопределяется метод `onAdvance()`. Конструктор класса `MyPhaser` получает один аргумент, который определяет количество выполняемых фаз. Обратите внимание на то, что класс `MyPhaser` автоматически регистрирует одну сторону. В данном примере это поведение полезно, но ваши приложения могут иметь другие потребности.

```
// Расширение класса Phaser и переопределение метода onAdvance() так,
// чтобы было выполнено только определенное количество фаз.

import java.util.concurrent.*;

// Расширение класса MyPhaser, чтобы позволить выполнять только
// определенное количество фаз.
class MyPhaser extends Phaser {
    int numPhases;

    MyPhaser(int parties, int phaseCount) {
        super(parties);
        numPhases = phaseCount - 1;
    }

    // Переопределить метод onAdvance(), чтобы выполнять
    // определенное количество фаз.
    protected boolean onAdvance(int p, int regParties) {
        // Этот оператор println() нужен только для иллюстрации.
        // Обычно метод onAdvance() не отображает вывод.
        System.out.println("Phase " + p + " completed.\n");

        // Если все фазы закончены, возвратить true
```

```
        if(p == numPhases || regParties == 0) return true;

        // В противном случае вернуть false.
        return false;
    }
}

class PhaserDemo2 {
    public static void main(String args[]) {

        MyPhaser phsr = new MyPhaser(1, 4);

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

        new MyThread(phsr, "A");
        new MyThread(phsr, "B");
        new MyThread(phsr, "C");

        // Ожидать завершения определенного количества фаз.
        while(!phsr.isTerminated()) {
            phsr.arriveAndAwaitAdvance();
        }

        System.out.println("The Phaser is terminated");
    }
}

// Поток выполнения, использующий Phaser.
class MyThread implements Runnable {
    Phaser phsr;
    String name;

    MyThread(Phaser p, String n) {
        phsr = p;
        name = n;
        phsr.register();
        new Thread(this).start();
    }

    public void run() {

        while(!phsr.isTerminated()) {
            System.out.println("Thread " + name + " Beginning Phase " +
                               phsr.getPhase());
            phsr.arriveAndAwaitAdvance();

            // Небольшая пауза для предотвращения перемешанного вывода.
            // Только для иллюстрации. Это не обязательно для правильной
            // работы фазера.
            try {
                Thread.sleep(10);
            } catch(InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}
```

Вывод представлен ниже.

```
Starting
Thread B Beginning Phase 0
Thread A Beginning Phase 0
Thread C Beginning Phase 0
```

```
Phase 0 completed.
Thread A Beginning Phase 1
Thread B Beginning Phase 1
Thread C Beginning Phase 1
Phase 1 completed.
Thread C Beginning Phase 2
Thread B Beginning Phase 2
Thread A Beginning Phase 2
Phase 2 completed.
Thread C Beginning Phase 3
Thread B Beginning Phase 3
Thread A Beginning Phase 3
Phase 3 completed.
The Phaser is terminated
```

Метод `main()` создает один экземпляр класса `Phaser`. В качестве аргумента ему передается значение 4, это значит, что он отработает четыре фазы, а затем остановится. Затем создаются три потока и начинается следующий цикл.

```
// Ожидать завершения определенного количества фаз.
while(!phsr.isTerminated()) {
    phsr.arriveAndAwaitAdvance();
}
```

Этот цикл просто вызывает метод `arriveAndAwaitAdvance()`, пока фазер не закончит работу, а он не закончит работу, пока не будет выполнено определенное количество фаз. В данном случае цикл продолжит выполнение до тех пор, пока не выполнятся четыре фазы. Обратите внимание на то, что потоки также вызывают метод `arriveAndAwaitAdvance()` в пределах выполняемого цикла, пока фазер не завершит работу. Это значит, что они выполняются до завершения определенного количества фаз.

Теперь внимательно рассмотрим код метода `onAdvance()`. Каждый раз, когда вызывается метод `onAdvance()`, передается текущая фаза и количество зарегистрированных сторон. Если текущая фаза соответствует указанной или количество зарегистрированных сторон равно нулю, метод `onAdvance()` возвращает значение `true`, останавливая таким образом фазер. Это осуществляет следующая строка кода.

```
// Если все фазы закончены, возвратить true
if(p == numPhases || regParties == 0) return true;
```

Как можно заметить, для достижения желаемого результата необходимо совсем не много кода.

Перед завершением темы имеет смысл указать, что вы не обязаны явно расширять класс `Phaser`. Как и в предыдущем примере, достаточно переопределить метод `onAdvance()`. В некоторых случаях может быть создан более компактный код при использовании анонимного внутреннего класса, переопределяющего метод `onAdvance()`.

У класса `Phaser` есть дополнительные возможности, которые могут быть полезны в ваших приложениях. Вы можете ожидать специфической фазы при вызове метода `awaitAdvance()`, как показано здесь.

```
int awaitAdvance(int фаза)
```

Здесь параметр *фаза* указывает номер фазы, на которой будет ожидать метод `awaitAdvance()`, пока не произойдет переход к следующей фазе. Он завершится немедленно, если аргумент, переданный в параметре *фаза*, не будет равен текущей фазе. Он также завершится немедленно, если фазер завершит работу. Однако если в параметре *фаза* будет передана текущая фаза, то он будет ждать до инкремента фазы.

Этот метод должен быть вызван только зарегистрированной стороной. Есть также прерывающая версия этого метода по имени `awaitAdvanceInterruptibly()`.

Чтобы зарегистрировать несколько сторон, вызовите метод `bulkRegister()`. Чтобы получить количество зарегистрированных сторон, вызовите метод `getRegisteredParties()`. Вы можете также получить количество сторон, завершивших и не завершивших работу, при вызове методов `getArrivedParties()` и `getUnarrivedParties()` соответственно. Чтобы перевести фазер в завершенное состояние, вызовите метод `forceTermination()`.

Класс `Phaser` позволяет также создать дерево фазеров. Он снабжен двумя дополнительными конструкторами, которые позволяют вам определять родителя и метод `getParent()`.

Использование исполнителя

Параллельный API поддерживает функциональную возможность, называемую *исполнителем* (executor), которая создает потоки и управляет ими. По сути, исполнитель предлагает альтернативный вариант управления потоками с помощью класса `Thread`.

В основе исполнителя лежит интерфейс `Executor`, который определяет следующий метод.

```
void execute(Runnable поток)
```

Выполняется поток, указанный в параметре *поток*. Таким образом, метод `execute()` запускает заданный поток.

Интерфейс `ExecutorService` расширяет интерфейс `Executor` за счет добавления методов, помогающих управлять выполнением потоков и контролировать его. Например, интерфейс `ExecutorService` определяет метод `shutdown()`, показанный ниже, который останавливает все потоки, находящиеся в данный момент под управлением экземпляра интерфейса `ExecutorService`.

```
void shutdown()
```

Интерфейс `ExecutorService` также определяет методы, которые запускают потоки, возвращающие результаты, выполняют совокупность потоков и определяют состояние останова. Некоторые из этих методов мы рассмотрим чуть позже.

Определяется также и интерфейс `ScheduledExecutorService`, который расширяет интерфейс `ExecutorService` для поддержки планирования потоков.

В параллельном API имеется три предварительно определенных класса исполнителей: `ThreadPoolExecutor`, `ScheduledThreadPoolExecutor` и `ForkJoinPool`. Класс `ThreadPoolExecutor` реализует интерфейсы `Executor` и `ExecutorService` и обеспечивает поддержку управляемого пула потоков. Класс `ScheduledThreadPoolExecutor` тоже реализует интерфейс `ScheduledExecutorService` для поддержки возможности планирования пула потоков. Класс `ForkJoinPool` реализует интерфейсы `Executor` и `ExecutorService`, он используется инфраструктурой Fork/Join Framework и рассматривается далее в этой главе.

Пул потоков предлагает набор потоков для решения разнообразных задач. Вместо того чтобы каждая задача имела дело со своим собственным потоком, используются потоки из пула. Это позволяет сократить нагрузку, связанную с созданием множества отдельных потоков. Хотя классы `ThreadPoolExecutor` и `ScheduledThreadPoolExecutor` можно использовать напрямую, чаще всего вам будет необходимо получать исполнителя при помощи вызова одного из следующих статических фабричных методов, определенных во вспомогательном классе `Executors`. Далее показаны некоторые примеры.

```
static ExecutorService newCachedThreadPool()
static ExecutorService newFixedThreadPool(int количПотоков)
static ScheduledExecutorService newScheduledThreadPool(int количПотоков)
```

Метод `newCachedThreadPool()` создает пул потоков, который не только добавляет потоки по мере необходимости, но и по возможности повторно их использует. Метод `newFixedThreadPool()` создает пул потоков, состоящий из определенного количества потоков. Метод `newScheduledThreadPool()` создает пул потоков, в котором можно осуществлять планирование потоков. Каждый из них возвращает ссылку на интерфейс `ExecutorService`, который можно использовать для управления пулом.

Простой пример исполнителя

Прежде чем продолжить обсуждение, давайте рассмотрим простой пример применения исполнителя. В следующей программе создается фиксированный пул, содержащий два потока. Затем этот пул используется для выполнения четырех задач. Таким образом, четыре задачи разделяют два потока, находящихся в пуле. После того как задачи будут выполнены, пул закрывается и программа завершает выполнение.

```
// Простой пример, в котором используется исполнитель.

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("Запуск");

        // Начало потоков.
        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("Завершение");
    }
}

class MyThread implements Runnable {
    String name;
    CountDownLatch latch;

    MyThread(CountDownLatch c, String n) {
        latch = c;
```

```
        name = n;

        new Thread(this);
    }

    public void run() {

        for(int i = 0; i < 5; i++) {
            System.out.println(name + ": " + i);
            latch.countDown();
        }
    }
}
```

Ниже показаны результаты выполнения программы.

```
Запуск
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
Завершение
```

Судя по результатам, несмотря на то что в пуле содержится всего два потока, выполняются все четыре задачи. Однако только две из них могут быть выполнены одновременно. Остальные должны ожидать, пока один из потоков пула не освободится, после чего его можно будет использовать.

Вызов метода `shutdown()` очень важен. Если бы его не было в программе, она не смогла бы завершиться, поскольку исполнитель оставался бы активным. Убедиться в этом можно, закомментировав вызов метода `shutdown()` и посмотрев, что из этого получится.

Использование интерфейсов `Callable` и `Future`

Одним из наиболее важных и, без сомнений, ярких средств в параллельном API является интерфейс `Callable`. Он представляет поток, возвращающий значение. Приложение может использовать объекты интерфейса `Callable` для вычисления результатов, которые затем будут возвращены вызывающему потоку. Это мощный механизм, поскольку он облегчает написание кода для множества различных числовых расчетов, в которых частичные результаты вычисляются одновременно. Его также можно использовать и для запуска потока, возвращающего код состояния, который свидетельствует об успешном выполнении потока.

Интерфейс Callable является обобщенным интерфейсом, который определяется следующим образом.

```
interface Callable<V>
```

Здесь параметр *V* задает тип данных, возвращаемых задачей. Интерфейс Callable определяет только один метод call().

```
V call() throws Exception
```

Внутри метода call() определяется задача, которую требуется выполнить. После того как она будет выполнена, возвращается результат. Если результат невозможно вычислить, метод call() передает исключение.

Задача интерфейса Callable решается при вызове метода submit(), определенного в интерфейсе ExecutorService. Метод submit() может иметь три формы, однако для интерфейса Callable используется только одна из них.

```
<T> Future<T> submit(Callable<T> задача)
```

Здесь параметр *задача* представляет объект интерфейса Callable, который будет выполняться в собственном потоке. Результат возвращается через объект интерфейса Future.

Интерфейс Future является обобщенным интерфейсом, представляющим значение, которое будет возвращено при помощи объекта интерфейса Callable. Поскольку это значение будет получено в некотором будущем, то имя интерфейса (Future) говорит само за себя. Интерфейс Future определяется следующим образом.

```
interface Future<V>
```

Здесь параметр *V* определяет тип результата.

Чтобы получить значение, нужно вызвать метод get() интерфейса Future, который имеет следующие две формы.

```
V get()
throws InterruptedException, ExecutionException
V get(long ожидать, TimeUnit tu) throws InterruptedException,
                                 ExecutionException, TimeoutException
```

В первом случае ожидание получения результатов длится бесконечно долго. Во втором случае можно указать период времени в параметре *ожидать*. Единицы периода времени в этом параметре задаются параметром *tu*, который представляет собой объект перечисления TimeUnit, рассматриваемый далее в этой главе.

В следующей программе показан пример применения интерфейсов Callable и Future. В ней будут созданы три задачи, выполняющие три разных вычисления. Первая задача возвращает суммарное значение, вторая находит длину гипотенузы прямоугольного треугольника с известными значениями длин его сторон, а третья определяет факториал для заданного значения. Все три вычисления производятся одновременно.

```
// Пример, в котором используется 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("Запуск");
```

```
        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("Завершение");
    }
}

// Три потока вычислений.
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;
    }
}
```

Ниже приводятся результаты выполнения программы.

```
Запуск
```

```
55
5.0
120
Завершение
```

Перечисление TimeUnit

Параллельный API определяет методы, принимающие параметр типа `TimeUnit`, который служит для определения периода времени. Перечисление `TimeUnit` используется для определения *временного разбиения* (или разрешения). Оно определено в пакете `java.util.concurrent` и может принимать одно из следующих значений.

- `DAYS`
- `HOURS`
- `MINUTES`
- `SECONDS`
- `MICROSECONDS`
- `MILLISECONDS`
- `NANOSECONDS`

Несмотря на то что с помощью перечисления `TimeUnit` можно определить любое из этих значений в вызовах методов, принимающих параметр синхронизации, нет гарантии того, что система сможет работать с заданным разрешением.

Ниже следует пример, в котором используется перечисление `TimeUnit`. Класс `CallableDemo`, показанный в предыдущем разделе, был изменен, чтобы использовать вторую форму метода `get()`, принимающего параметр типа `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 этом варианте ни один из вызовов метода `get()` не будет ожидать дольше 10 миллисекунд.

Перечисление `TimeUnit` определяет различные методы, выполняющие преобразование единиц.

```
long convert(long tзнач, TimeUnit tu)
long toMicros(long tзнач)
long toMillis(long tзнач)
long toNanos(long tзнач)
long toSeconds(long tзнач)
long toDays(long tзнач)
long toHours(long tзнач)
long toMinutes(long tзнач)
```

Метод `convert()` преобразует *`tзнач`* в определенные единицы и возвращает результат. Методы `toXXX` выполняют указанное преобразование и возвращают результат.

Перечисление `TimeUnit` также определяет следующие методы синхронизации.

```
void sleep(long задержка) throws InterruptedExecution
void timedJoin(Thread поток, long задержка) throws InterruptedExecution
void timedWait(Object объект, long задержка) throws InterruptedExecution
```

Метод `sleep()` приостанавливает выполнение на определенный период времени, который задается в виде вызывающей константы перечисления. Он преобразуется в вызов метода `Thread.sleep()`.

Метод `timedJoin()` является специализированной версией метода `Thread.join()`, в котором поток (параметр *поток*) приостанавливается на период времени, указанный параметром *задержка*. Метод `timedWait()` является специализированной версией метода `Object.wait()`, в котором *объект* ожидает период времени, заданный параметром *задержка*, который исчисляется в вызывающих единицах времени.

Параллельные коллекции

Как уже отмечалось, параллельный API определяет несколько коллекций, которые были разработаны для выполнения параллельных операций. К ним относятся следующие коллекции.

- `ArrayBlockingQueue`
- `ConcurrentHashMap`
- `ConcurrentLinkedDeque` (Добавлено в JDK 7)
- `ConcurrentLinkedQueue`
- `ConcurrentSkipListMap`
- `ConcurrentSkipListSet`
- `CopyOnWriteArrayList`
- `CopyOnWriteArraySet`
- `DelayQueue`
- `LinkedBlockingDeque`
- `LinkedBlockingQueue`
- `LinkedTransferQueue` (Добавлено в JDK 7)
- `PriorityBlockingQueue`
- `SynchronousQueue`

Они предлагают параллельные альтернативы связанным с ними классам, определенным в инфраструктуре Collections Framework. Эти коллекции работают подобно другим коллекциям, за исключением того, что они поддерживают параллелизм. Программисты, знакомые с инфраструктурой Collections Framework, не будут иметь проблем с использованием этих параллельных коллекций.

Блокировки

Пакет `java.util.concurrent.locks` предоставляет поддержку *блокировок* (lock), которые являются объектами, предлагающими альтернативу использованию блоков `synchronized` для управления доступом к общему ресурсу. Давайте разберемся с тем, как работают блокировки. Прежде чем получить доступ к общему ре-

сурсу, запрашивается блокировка, защищающая этот ресурс. Когда доступ к ресурсу будет завершен, блокировка снимается. Если второй поток попытается запросить блокировку в тот момент, когда она используется еще каким-нибудь потоком, первый поток будет ожидать, пока блокировка не будет снята. Благодаря этому появляется возможность избежать возникновения конфликта доступа к общему ресурсу.

Блокировки особенно полезны тогда, когда нескольким потокам нужно получить доступ к значению из общих данных. Например, приложение складского учета может иметь поток, который сначала подтверждает, что товар имеется на складе, а затем уменьшает количество доступных товаров после каждой продажи. Если будет выполняться два или более таких потока, то без синхронизации может получиться так, что один поток начнет свою транзакцию в момент выполнения транзакции другим потоком. В результате этого оба потока будут предполагать о существовании достаточного количества товара, хотя на самом деле товара будет ровно столько, сколько требуется для осуществления одной продажи. В подобных ситуациях с помощью блокировок можно организовывать синхронную работу потоков.

Каждая блокировка реализует интерфейс `Lock`. В табл. 27.2 перечислены методы, определенные интерфейсом `Lock`. В общем случае для запроса блокировки необходимо вызвать метод `lock()`. Если блокировка не будет доступна, метод `lock()` войдет в режим ожидания. Для снятия блокировки вызовите метод `unlock()`. Чтобы узнать, является ли блокировка свободной, и запросить ее, если она свободна, вызовите метод `tryLock()`. Метод не будет ожидать блокировку, если она не является доступной. Наоборот, он возвращает значение `true`, если блокировка получена, и значение `false` – если нет. Метод `newCondition()` возвращает объект `Condition`, связанный с блокировкой. Применение `Condition` позволяет расширить возможности управления блокировками с помощью методов `await()` и `signal()`, которые обеспечивают функциональные возможности, подобные тем, что предлагаются методами `Object.wait()` и `Object.notify()`.

Таблица 27.2. Методы интерфейса `Lock`

Метод	Описание
`void lock()`	Ожидание длится до тех пор, пока вызываемая блокировка не может быть получена
`void lockInterruptibly() throws InterruptedException`	Ожидание длится до тех пор, пока вызываемая блокировка не может быть получена, если только не произойдет прерывание
`Condition newCondition()`	Возвращает объект `Condition`, связанный с вызываемой блокировкой
`boolean tryLock()`	Пытается запросить блокировку. Этот метод не входит в режим ожидания, если блокировка не является свободной. Вместо этого он возвращает значение `true`, если блокировка была получена, и значение `false`, если на данный момент блокировка используется другим потоком
`boolean tryLock(long` *`ожидать`*`, TimeUnit` *`tu`*`) throws InterruptedException`	Пытается получить блокировку. Если блокировка недоступна, то метод будет ожидать столько времени, сколько указано в параметре *`ожидать`*, единицы которого определены параметром *`tu`*. Он возвращает значение `true`, если блокировка была получена, и значение `false`, если блокировка не была получена в течение заданного периода
`void unlock()`	Снимает блокировку

Пакет java.util.concurrent.locks содержит реализацию интерфейсов Lock и ReentrantLock, которые реализуют *реентерабельную блокировку*, представляющую собой блокировку, в которую поток, удерживающий на данный момент эту блокировку, может входить повторно. (Естественно, если поток входит в блокировку повторно, все вызовы метода lock() должны быть смещены на равное количество вызовов метода unlock().) В противном случае поток, пытающийся запросить блокировку, перейдет в режим ожидания до тех пор, пока она не будет освобождена.

В следующей программе демонстрируется пример использования блокировок. В ней создается два потока, которые обращаются к общему ресурсу, переменной Shared.count. Прежде чем поток сможет обратиться к переменной Shared.count, он должен получить блокировку. После получения блокировки значение переменной Shared.count увеличивается, после чего, не снимая блокировки, поток входит в режим простоя. Вследствие этого другой поток будет пытаться получить блокировку. Однако поскольку блокировка все еще удерживается первым потоком, другой поток будет ожидать до тех пор, пока первый поток не выйдет из режима простоя и не снимет блокировку. Результаты выполнения показывают, что доступ к переменной Shared.count синхронизируется с помощью блокировки.

```
// Простой пример блокировки.

import java.util.concurrent.locks.*;

class LockDemo {

    public static void main(String args[]) {
        ReentrantLock lock = new ReentrantLock();

        new LockThread(lock, "A");
        new LockThread(lock, "B");
    }
}

// Общий ресурс.
class Shared {
    static int count = 0;
}

// Поток выполнения, увеличивающий значение счета.
class LockThread implements Runnable {
    String name;
    ReentrantLock lock;
    LockThread(ReentrantLock lk, String n) {
        lock = lk;
        name = n;
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Запуск " + name);

        try {
            // Сначала блокируется счетчик.
            System.out.println(name + " ожидает блокирования счетчика.");
            lock.lock();
            System.out.println(name + " блокирует счетчик.");
            Shared.count++;
            System.out.println(name + ": " + Shared.count);
            // Теперь, если это возможно, разрешается контекстное
```

```
            // переключение.
            System.out.println(name + " простаивает.");
            Thread.sleep(1000);
        } catch (InterruptedException exc) {
            System.out.println(exc);
        } finally {
            // Снятие блокировки
            System.out.println(name + " разблокирует счетчик.");
            lock.unlock();
        }
    }
}
```

Ниже показаны результаты выполнения. (У вас порядок выполнения потоков может быть другим.)

```
Запуск A
A ожидает блокирования счетчика.
A блокирует счетчик.
A: 1
A простаивает.
Запуск B
B ожидает блокирования счетчика.
A разблокирует счетчик.
B блокирует счетчик.
B: 2
B простаивает.
B разблокирует счетчик.
```

Пакет `java.util.concurrent.locks` также реализует интерфейс `ReadWriteLock`, определяющий реентерабельную блокировку, которая поддерживает отдельные блокировки для доступа на чтение и запись. Это позволит предоставлять читателям ресурса несколько блокировок до его записи. Класс `ReentrantReadWriteLock` предлагает реализацию интерфейса `ReadWriteLock`.

Атомарные операции

Пакет `java.util.concurrent.atomic` предлагает альтернативный вариант другим функциональным возможностям синхронизации при чтении или записи значения некоторых типов переменных. В этом пакете доступны методы, которые получают, задают или сравнивают значение переменной во время одной непрерывной (т.е. атомарной) операции. А это значит, что теперь не будет нужна ни блокировка, ни любой другой механизм синхронизации.

Атомарные операции выполняются с помощью классов `AtomicInteger` и `AtomicLomg`, а также методов `get()`, `set()`, `compareAndSet()`, `decrementAndGet()` и `getAndSet()`, которые реализуют действия, соответствующие их именам.

Ниже показан пример, показывающий, как можно синхронизировать доступ к общему ресурсу с помощью класса `AtomicInteger`.

```
// Простой пример атомарных операций.

import java.util.concurrent.atomic.*;

class AtomicDemo {
    public static void main(String args[]) {
        new AtomThread("A");
        new AtomThread("B");
        new AtomThread("C");
```

```
    }
}

class Shared {
    static AtomicInteger ai = new AtomicInteger(0);
}

// Поток выполнения, при котором увеличивается значение счета.
class AtomThread implements Runnable {
    String name;
    AtomThread(String n) {
        name = n;
        new Thread(this).start();
    }

    public void run() {
        System.out.println("Запуск " + name);
        for(int i=1; i <= 3; i++)
            System.out.println(name + " получено: " +
                               Shared.ai.getAndSet(i));
    }
}
```

В этой программе при помощи класса `Shared` создается статический объект `ai` класса `AtomicInteger`. Затем создаются три потока класса `AtomThread`. В методе `run()` объект `Shared.ai` изменяется вызовом метода `getAndSet()`. Этот метод возвращает предыдущее значение и устанавливает то значение, которое было передано в качестве параметра. Благодаря классу `AtomicInteger` исключается вероятность того, что два потока будут одновременно осуществлять запись в объект `ai`.

В общем случае атомарные операции предлагают удобную (а, возможно, и более эффективную) альтернативу другим механизмам синхронизации при работе с одной переменной.

Параллельное программирование при помощи инфраструктуры Fork/Join Framework

В последние годы появилась новая важная тенденция в разработке программного обеспечения — *параллельное программирование* (parallel programming). Параллельное программирование — это общее название технологий, которые используют в своих интересах многоядерные процессоры, содержащие два или больше ядер. Как известно большинству читателей, ныне многоядерные компьютеры — вполне обычное явление. Преимуществом многопроцессорных систем является возможность значительного увеличения производительности программы. В результате возросла потребность в механизме, который предоставит программистам Java простой, но эффективный способ использования нескольких процессоров ясным и масштабируемым способом. В ответ на эту потребность в комплект JDK 7 было добавлено несколько новых классов и интерфейсов поддержки параллельного программирования. Обычно они упоминаются как инфраструктура Fork/Join Framework. Это одно из наиболее важных добавлений, внесенных комплектом JDK 7 в библиотеку классов Java. Инфраструктура Fork/Join Framework определена в пакете `java.util.concurrent`.

Инфраструктура Fork/Join Framework улучшает многопоточное программирование двумя важными способами. Во-первых, она упрощает создание и использование нескольких потоков. Во-вторых, это автоматизирует использование несколь-

ких процессоров. Другими словами, при использовании инфраструктуры Fork/Join Framework вы позволяете вашим приложениям автоматически масштабировать количество доступных для использования процессоров. Эти две возможности позволяют считать инфраструктуру Fork/Join Framework рекомендованным подходом многопоточного программирования, когда желательна параллельная обработка.

Прежде чем продолжить, следует указать на различие между традиционным многопоточным и параллельным программированием. В прошлом большинство компьютеров имело один процессор, и многопоточность применялась, прежде всего, для использования периодов ожидания, таких как при ожидании программой пользовательского ввода. При этом подходе один поток может выполняться, в то время как другой ожидает. Другими словами, в системе с одним процессором многопоточность используется для того, чтобы позволить двум или более задачам совместно использовать процессор. Этот тип многопоточности, как правило, поддерживается объектом класса `Thread` (как описано в главе 11). Хотя этот тип многопоточности будет оставаться весьма полезным всегда, он не был оптимизирован для ситуаций, когда у вас есть два или больше процессоров (многоядерный компьютер).

Когда доступно несколько процессоров, необходим другой тип многопоточности, который обеспечивает истинное параллельное выполнение. С двумя или больше процессорами можно выполнять части программы одновременно — каждая часть выполняется на собственном процессоре. Это применяется для значительного ускорения выполнения операций некоторых типов, таких как сортировка, преобразование или поиск в большом массиве. Во многих случаях такие операции могут быть разделены на меньшие части (каждая из них воздействует на часть массива), и каждая часть может быть запущена на собственном процессоре. Как можно догадаться, выигрыш в эффективности может быть огромным. Просто представьте: в будущем каждый программист будет применять параллельное программирование, поскольку это открывает путь к существенному повышению производительности программы.

Основные классы инфраструктуры Fork/Join Framework

Инфраструктура Fork/Join Framework расположена в пакете `java.util.concurrent`. Ее ядро составляют следующие классы.

`ForkJoinTask<V>`	Абстрактный класс, определяющий задачу
`ForkJoinPool`	Управляет выполнением объекта класса `ForkJoinTask`
`RecursiveAction`	Производный от класса `ForkJoinTask<V>` класс для задач, которые не возвращают значений
`RecursiveTask<V>`	Производный от класса `ForkJoinTask<V>` класс для задач, возвращающих значения

Рассмотрим отношения между ними. Класс `ForkJoinPool` управляет выполнением объекта класса `ForkJoinTask`. Класс `ForkJoinTask` — это абстрактный класс, который расширяется двумя другими абстрактными классами: `RecursiveAction` и `RecursiveTask`. Как правило, ваш код расширяет эти классы, чтобы создать задачу. Прежде чем перейти к подробному рассмотрению процесса, сделаем краткий обзор ключевых аспектов каждого класса.

Класс `ForkJoinTask<V>`

Класс `ForkJoinTask<V>` является абстрактным и определяет задачу, которой может управлять объект класса `ForkJoinPool`. Параметр типа `V` опреде-

ляет тип результата задачи. Класс ForkJoinTask отличается от класса Thread тем, что представляет облегченную абстракцию задачи, а не поток выполнения. Класс ForkJoinTask выполняется потоками, управляемыми пулом потока класса ForkJoinPool. Этот механизм позволяет управлять большим количеством задач небольшим количеством фактических потоков. Таким образом, класс ForkJoinTask весьма эффективен, по сравнению с потоками.

В классе ForkJoinTask определено много методов. Основные, fork() и join(), представлены ниже.

```
final ForkJoinTask<V> fork()
final V join()
```

Метод fork() передает вызывающую задачу для асинхронного выполнения. Это значит, что поток, который вызывает метод fork(), продолжает выполняться. После того как задача запланирована для выполнения, метод fork() возвращает this. Это может быть сделано только внутри вычислительной части другого объекта класса ForkJoinTask, который выполняется в пределах объекта класса ForkJoinPool. (Вскоре вы узнаете, как это делается.) Метод join() ожидает завершения задачи, для которой он вызван. Возвращается результат задачи. Таким образом, с помощью методов fork() и join() вы можете запустить одну или несколько новых задач, а затем ждать их завершения.

Другой важный метод класса ForkJoinTask — это метод invoke(). Он объединяет операции ветвления и объединения в единый вызов, поскольку запускает задачу, а затем ждет ее завершения. Это продемонстрировано здесь.

```
final V invoke()
```

Возвращается результат вызывающей задачи.

При помощи метода invokeAll() вы можете вызвать несколько задач за один раз. Две его формы представлены здесь.

```
static void invokeAll(ForkJoinTask<?> задачаА, ForkJoinTask<?> задачаВ)
static void invokeAll(ForkJoinTask<?> ... списокЗадач)
```

В первом случае выполняются задачи *задачаА* и *задачаВ*, во втором случае — все определенные задачи. В обоих случаях вызывающий поток ожидает завершения всех определенных задач. Метод invokeAll() может быть вызван только внутри вычислительной части другого объекта класса ForkJoinTask, который выполняется в пределах объекта класса ForkJoinPool.

Класс RecursiveAction

Этот класс происходит от класса ForkJoinTask и инкапсулирует задачу, которая не возвращает результат. Как правило, ваш код расширяет класс RecursiveAction, чтобы создать задачу, типом возвращаемого значения которого является void. В классе RecursiveAction определено четыре метода, но только один обычно представляет интерес — абстрактный метод по имени compute(). Когда вы расширите класс RecursiveAction, чтобы создать реальный класс, помещаете код, который определяет задачу, в метод compute(). Метод compute() представляет *вычислительную* (computational) часть задачи.

Класс RecursiveAction определяет метод compute() так.

```
protected abstract void compute()
```

Обратите внимание на то, что метод compute() защищенный. Это означает, что он может быть вызван только другими методами данного класса или класса, производного от него. Кроме того, поскольку метод абстрактный, его следует реализовать в производном классе (если этот производный класс тоже не абстрактный).

Как правило, класс RecursiveAction используется для реализации рекурсивной стратегии для задач, которые не возвращают результатов. (См. раздел "Стратегия «разделяй и властвуй»" далее в этой главе.)

Класс RecursiveTask<V>

Еще одним производным от класса ForkJoinTask является класс RecursiveTask<*V*>. Данный класс инкапсулирует задачу, которая возвращает результат. Тип результата определяет параметр *V*. Как правило, ваш код расширит класс RecursiveTask<*V*>, чтобы создать задачу, которая возвращает значение. Как и в классе RecursiveAction, здесь определено четыре метода, но обычно используется только абстрактный метод compute(), который представляет вычислительную часть задачи. Когда вы расширяете класс RecursiveTask<*V*> для создания реального класса, в метод compute() помещают код, который представляет задачу. Этот код также должен возвратить результат задачи.

Класс RecursiveTask<*V*> определяет метод compute() так.

```
protected abstract V compute()
```

Обратите внимание на то, что метод compute() защищенный. Это означает, что он может быть вызван только другими методами данного класса или класса, производного от него. Кроме того, поскольку метод абстрактный, его следует реализовать в производном классе. Будучи реализованным, он должен возвращать результат задачи.

Как правило, класс RecursiveTask используется при реализации рекурсивной стратегии для задач, которые возвращают результат. (См. раздел "Стратегия «разделяй и властвуй»" далее в этой главе.)

Класс ForkJoinPool

Выполнение объекта класса ForkJoinTask происходит в пределах объекта класса ForkJoinPool, который управляет также выполнением задач. Поэтому, чтобы запустить объект класса ForkJoinTask, сначала необходим объект класса ForkJoinPool.

В классе ForkJoinPool определено несколько конструкторов. Вот два наиболее популярных.

```
ForkJoinPool()
ForkJoinPool(int уровеньПаралл)
```

Первый создает стандартный пул, обеспечивающий уровень параллелизма, равный количеству процессоров, доступных в системе. Второй позволяет задать уровень параллелизма. Его значение должно быть больше нуля и не больше предела для реализации. Уровень параллелизма определяет количество потоков, которые могут выполняться одновременно. В результате уровень параллелизма фактически определяет количество задач, которые могут выполняться одновременно. (Конечно, количество одновременно выполняемых задач не может превышать количество процессоров.) Следует однако уяснить, что уровень параллелизма *не ограничивает* количество задач, которыми может управлять пул. Объект класса ForkJoinPool может управлять существенно большим количеством задач, чем его уровень параллелизма. Кроме того, уровень параллелизма — это только цель, а не гарантия.

После того как вы создали экземпляр класса ForkJoinPool, можете запустить задачу многими способами. Задачу, запущенную первой, обычно называют основной. Эта задача нередко запускает подчиненные задачи, которыми также управляет пул. Наиболее распространенный способ запуска основной задачи подразумевает вызов метода invoke() класса ForkJoinPool, как показано далее.

```
<T> T invoke(ForkJoinTask<T> задача)
```

Этот метод запускает задачу, определенную параметром *задача*, и возвращает результат ее выполнения. Это значит, что вызывающий код ожидает завершения метода `invoke()`.

Чтобы запустить задачу и не ждать ее завершения, вы можете использовать метод `execute()`. Вот одна из его форм.

```
void execute(ForkJoinTask<?> задача)
```

В данном случае *задача* запускается, но вызывающий код не ждет ее завершения. Вместо этого вызывающий код продолжает выполнение асинхронно.

Класс `ForkJoinPool` управляет выполнением своих потоков, используя подход под названием *захват задачи* (work-stealing). Каждый рабочий поток поддерживает очередь задач. Если очередь одного рабочего потока окажется пуста, то она возьмет задачу от другого рабочего потока. Это способствует повышению общей производительности и помогает поддерживать баланс нагрузки. (Из-за запросов процессорного времени другими процессами в системе, даже два рабочих потока с одинаковыми задачами в их очередях не могут завершиться одновременно.)

Еще один момент: класс `ForkJoinPool` использует *потоки-демоны* (daemon thread). Поток-демон автоматически заканчивается, когда заканчиваются все пользовательские потоки. Таким образом, нет никакой необходимости явно завершить работу объекта класса `ForkJoinPool`. Однако это можно сделать при вызове метода `shutdown()`.

Стратегия "разделяй и властвуй"

Как правило, инфраструктура Fork/Join Framework применяет *стратегию "разделяй и властвуй"*, лежащую в основе рекурсии. Вот почему два класса, производных от класса `ForkJoinTask`, называются `RecursiveAction` и `RecursiveTask`. Ожидается, что вы расширите один из этих классов при создании собственной задачи ветвления/объединения.

Стратегия "разделяй и властвуй", лежащая в основе рекурсии, подразумевает разделение задачи на подзадачи, пока их размер не станет достаточно маленьким для последовательной обработки. Например, задача, которая применяет преобразование к каждому из N элементов в массиве целых чисел, может быть разделена на две подзадачи, каждая из которых преобразует половину элементов в массиве. Таким образом, одна подзадача преобразует элементы от 0 до $N/2$, а другая — элементы от $N/2$ до N. В свою очередь, каждая подзадача может быть сведена к набору подзадач, каждая из которых преобразует половину остальных элементов. Этот процесс деления массива продолжится до тех пор, пока не будет достигнуто пороговое значение, при котором последовательное решение оказывается быстрее, чем создание следующего разделения.

Преимущество стратегии "разделяй и властвуй" заключается в том, что обработка может осуществляться параллельно. Поэтому, вместо того чтобы циклически перебирать весь массив, используя один поток, части массива могут быть обработаны одновременно. Конечно, подход "разделяй и властвуй" работает во многих ситуациях и без массива (или коллекции), но наиболее распространенная область применения подразумевает некоторый тип массива, коллекции или группы данных.

Одним из ключевых моментов наилучшего использования стратегии "разделяй и властвуй" является правильное определение порогового значения, после которого используется последовательная обработка (а не дальнейшее деление). Как правило, оптимальное пороговое значение получается при профилировании характеристик выполнения. Однако даже при использовании порогового значения меньше оптимального, все равно произойдет весьма существенное ускоре-

ние. Однако лучше избегать чрезмерно больших или маленьких пороговых значений. На момент написания этой книги документация по API Java для метода `ForkJoinTask<T>` утверждает, что эмпирически выведено правило, согласно которому задача должна выполняться где-то за 100–10 000 этапов вычисления.

Важно также понимать, что на оптимальное пороговое значение влияет также период времени, занимаемый вычислением. Если каждый этап вычисления достаточно продолжителен, то предпочтительнее меньшие пороговые значения, и, наоборот, если каждый вычислительный этап очень короткий, то большие пороговые значения могут обеспечить лучшие результаты. Для приложений, которые должны быть запущены на известной системе с известным количеством процессоров, вы можете использовать информацию о количестве процессоров для обоснования решения о пороговом значении. Но для приложений, которые будут выполняться на множестве систем, возможности которых не известны заранее, вы не можете сделать предположение о среде выполнения.

Еще один момент: хотя в системе может быть доступно несколько процессоров, другие задачи (и сама операционная система) будут конкурировать с вашим приложением за процессорное время. Таким образом, не стоит полагать, что у вашей программы будет неограниченный доступ ко всем процессорам. Кроме того, разные процессы той же программы могут показать разные характеристики времени выполнения из-за различий в нагрузке.

Первый простой пример ветвления/объединения

Теперь рассмотрим простой пример, демонстрирующий инфраструктуру Fork/Join Framework и стратегию "разделяй и властвуй" в действии. Вот программа, которая преобразует элементы массива типа `double` в их квадратные корни. Для этого используется производный класс `RecursiveAction`.

```
// Простой пример фундаментальной стратегии "разделяй и властвуй".
// В данном случае используется класс RecursiveAction.
import java.util.concurrent.*;
import java.util.*;

// Класс ForkJoinTask (через RecursiveAction) преобразует элементы
// массива double в их квадратные корни.
class SqrtTransform extends RecursiveAction {
    // В этом примере пороговое значение произвольно устанавливается
    // равным 1 000. В реальном коде его оптимальное значение может
    // быть определено при профилировании и экспериментально.
    final int seqThreshold = 1000;

    // Обрабатываемый массив.
    double[] data;

    // Определяет часть обрабатываемых данных.
    int start, end;

    SqrtTransform(double[] vals, int s, int e ) {
        data = vals;
        start = s;
        end = e;
    }
    // Этот метод осуществляет параллельное вычисление.
    protected void compute() {

        // Если количество элементов ниже порогового значения, то
```

```
        // выполнять обработку последовательно.
        if((end - start) < seqThreshold) {
            // Преобразовать 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 середину.
            int middle = (start + end) / 2;

            // Запустить новые задачи, используя 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 некие значения.
        for(int i = 0; i < nums.length; i++)
            nums[i] = (double) i;

        System.out.println("A portion of the original sequence:");

        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 главный ForkJoinTask.
        fjp.invoke(task);

        System.out.println("A portion of the transformed sequence" +
                           " (to four decimal places):");

        for(int i=0; i < 10; i++)
            System.out.format("%.4f ", nums[i]);
        System.out.println();
    }
}
```

Вот вывод программы.

```
A portion of the original sequence:
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

A portion of the transformed sequence (to four decimal places):
0.0000 1.0000 1.4142 1.7321 2.0000 2.2361 2.4495 2.6458 2.8284 3.0000
```

Как можно заметить, значения элементов массива были преобразованы в их квадратные корни.

Давайте внимательнее рассмотрим работу этой программы. В первую очередь обратите внимание на то, что класс `SqrtTransform` расширяет класс `RecursiveAction`. Как упоминалось, класс `RecursiveAction` расширяет класс `ForkJoinTask` для задач, которые не возвращают результатов. Затем обратите внимание на финальную переменную `seqThreshold`. Ее значение определяет, когда будет иметь место последовательная обработка. Это значение устанавливается (несколько произвольно) равным 1 000. Далее обратите внимание на то, что ссылка на обрабатываемый массив сохраняется в переменной `data` и что поля `start` и `end` используются для указания границ элементов для обращения.

Основное действие программы осуществляется в методе `compute()`. Все начинается с проверки количества подлежащих обработке элементов на предмет соответствия нижнему пороговому значению для последовательной обработки. Если это так, то эти элементы обрабатываются (в данном примере вычисляется их квадратный корень). Если пороговое значение для последовательной обработки не достигнуто, то вызов метода `invokeAll()` запускает две новые задачи. В данном случае каждая подзадача обрабатывает половину элементов. Как объяснялось ранее, метод `invokeAll()` ожидает завершения обеих задач. После завершения всех рекурсивных вызовов каждый элемент в массиве будет изменен, причем большая частью действий осуществляется параллельно (если доступно несколько процессоров).

Влияние уровня параллелизма

Прежде чем завершить тему, важно рассмотреть влияние уровня параллелизма на эффективность выполнения задачи ветвления/объединения, а также взаимосвязь между параллелизмом и пороговым значением. Программа, приведенная в этом разделе, позволяет вам экспериментировать с различными уровнями параллелизма и пороговыми значениями. С учетом использования многоядерного компьютера, вы сможете в интерактивном режиме наблюдать результат изменения этих значений.

В предыдущем примере, поскольку там использовался стандартный конструктор класса `ForkJoinPool`, по умолчанию был задан уровень параллелизма, равный количеству процессоров в системе. Однако вы можете задать уровень параллелизма по своему желанию. Один из продемонстрированных ранее способов подразумевает его определение при создании объекта класса `ForkJoinPool` с использованием следующего конструктора.

```
ForkJoinPool(int уровеньПаралл)
```

Здесь параметр *уровеньПаралл* определяет уровень параллелизма. Его значение должно быть больше нуля, но меньше предела, определенного реализацией.

Следующая программа создает задачу ветвления/объединения, которая преобразует массив типа `double`. Преобразование произвольно, но оно задумано так, чтобы использовалось несколько циклов процессора. Это сделано для большей наглядности отображения эффекта от изменения порогового значения или уровня параллелизма. При использовании программы укажите пороговое значение и уровень параллелизма в командной строке. Затем программа запускает задачи. Она отображает также период времени, занятый выполнением задачи. Для этого используется метод `System.nanoTime()`, возвращающий значение высокоточного таймера JVM.

```
// Пример программы, позволяющей экспериментировать с эффектом от
// изменения порогового значения и параллелизма класса ForkJoinTask.
import java.util.concurrent.*;
```

```
// Класс ForkJoinTask (через RecursiveAction) преобразует
// элементы массива double.
class Transform extends RecursiveAction {

    // Порог последовательного выполнения, устанавливаемый конструктором.
    int seqThreshold;

    // Обрабатываемый массив.
    double[] data;

    // Определяет часть обрабатываемых данных.
    int start, end;

    Transform(double[] vals, int s, int e, int t ) {
        data = vals;
        start = s;
        end = e;
        seqThreshold = t;
    }

    // Этот метод осуществляет параллельное вычисление.
    protected void compute() {

        // Если количество элементов ниже порогового значения, то
        // выполнять обработку последовательно.
        if((end - start) < seqThreshold) {
            // Следующий код присваивает элементу по четному индексу
            // квадратный корень его первоначального значения. Элементу
            // по нечетному индексу присваивается кубический корень его
            // первоначального значения.
            // Этот код предназначен только для использования
            // процессорного времени, чтобы эффект от параллельного
            // выполнения был нагляднее.
            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 середину.
            int middle = (start + end) / 2;

            // Запустить новые задачи, используя 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("Usage: FJExperiment parallelism
                               threshold ");
            return;
        }

        pLevel = Integer.parseInt(args[0]);
        threshold = Integer.parseInt(args[1]);

        // Эти переменные используются для замера времени 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);

        // Старт хронометража.
        beginT = System.nanoTime();

        // Запустить главный ForkJoinTask.
        fjp.invoke(task);

        // Конец хронометража.
        endT = System.nanoTime();

        System.out.println("Level of parallelism: " + pLevel);
        System.out.println("Sequential threshold: " + threshold);
        System.out.println("Elapsed time: " + (endT - beginT) + " ns");
        System.out.println();
    }
}
```

Чтобы использовать программу, укажите уровень параллелизма, а затем пороговое значение. Вы можете экспериментально опробовать различные значения для каждого параметра и понаблюдать за результатами. Помните, вы должны запускать код на компьютере, по крайней мере, с двумя процессорами. Кроме того, выполнение на двух разных компьютерах почти наверняка приведет к разным результатам, из-за наличия в системе других процессов, использующих процессорное время.

Чтобы получить общее представление о значении, которое имеет параллелизм, проделайте такой эксперимент. Сначала запустите программу так.

```
java FJExperiment 1 1000
```

Это запросит уровень параллелизма 1 (совершенно последовательное выполнение) при пороговом значении 1000. Вот пример выполнения, полученный на двухъядерном компьютере.

```
Level of parallelism: 1
Sequential threshold: 1000
Elapsed time: 259677487 ns
```

Теперь укажите уровень параллелизма 2 так.

```
java FJExperiment 2 1000
```

Вот пример вывода, полученный на том же двухъядерном компьютере.

```
Level of parallelism: 2
Sequential threshold: 1000
Elapsed time: 169254472 ns
```

Вполне очевидно, что применение параллелизма существенно уменьшает время выполнения, увеличивая таким образом скорость программы. Поэкспериментируйте с изменением порогового значения и параллелизма на собственном компьютере. Результаты могут удивить вас.

Есть еще два метода, которые вы могли бы найти полезными при экспериментировании с характеристиками выполнения программы ветвления/объединения. Во-первых, вы можете получить уровень параллелизма при вызове метода `getParallelism()`, определенного в классе `ForkJoinPool`. Вот его форма.

```
int getParallelism()
```

Он возвращает текущий уровень параллелизма. Напомню, что по умолчанию он будет равен количеству доступных процессоров. Во-вторых, вы можете получить количество процессоров, доступных в системе, при вызове метода `availableProcessors()`, который определяется классом `Runtime`. Вот его форма.

```
int availableProcessors()
```

В связи с наличием других системных запросов, возвращаемое значение может изменяться от вызова к вызову.

Пример применения класса `RecursiveTask<V>`

Два приведенных выше примера основаны на классе `RecursiveAction`, и это значит, что они одновременно выполняют задачи, которые не возвращают результатов. Чтобы создать задачу, которая возвращает результат, используйте класс `RecursiveTask`. Сами решения разработаны так же, как и в предыдущем примере. Основное отличие в том, что метод `compute()` возвращает результат. Таким образом, следует объединить результаты так, чтобы при завершении первого запроса он возвращал общий результат. Другое отличие в том, что обычно вы будете запускать подзадачу при вызове метода `fork()` или `join()` явно (а не неявно, при вызове метода `invokeAll()`, например).

Следующая программа демонстрирует применение метода `RecursiveTask`. Она создает задачу по имени `Sum`, которая возвращает сумму значений в массиве типа `double`. В этом примере массив состоит из 5 000 элементов. Однако каждое второе значение отрицательное. Таким образом, первыми значениями массива будут 0, –1, 2, –3, 4 и т.д.

```
// Простой пример использования RecursiveTask<V>.
import java.util.concurrent.*;

// Класс RecursiveTask, используемый для вычисления суммы массива
// типа double.
class Sum extends RecursiveTask<Double> {

    // Пороговое значение последовательного выполнения.
    final int seqThresHold = 500;

    // Обрабатываемый массив.
    double[] data;

    // Определяет часть обрабатываемых данных.
    int start, end;
```

```
    Sum(double[] vals, int s, int e ) {
        data = vals;
        start = s;
        end = e;
    }

    // Поиск суммы массива типа double.
    protected Double compute() {
        double sum = 0;

        // Если количество элементов ниже порогового значения, то
        // выполнять обработку последовательно.
        if((end - start) < seqThresHold) {
            // Сумма элементов.
            for(int i = start; i < end; i++) sum += data[i];
        }
        else {
            // В противном случае продолжить разделение данных на
            // меньшие части.

            // Найти середину.
            int middle = (start + end) / 2;

            // Запустить новые задачи, используя разделенные на
            // части данные.
            Sum subTaskA = new Sum(data, start, middle);
            Sum subTaskB = new Sum(data, middle, end);

            // Запустить каждую подзадачу при разветвлении.
            subTaskA.fork();
            subTaskB.fork();

            // Подождать завершения подзадач и агрегировать результаты.
            sum = subTaskA.join() + subTaskB.join();
        }
        // Вернуть конечную 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 значениями, знаки которых чередуются.
        for(int i=0; i < nums.length; i++)
            nums[i] = (double) (((i%2) == 0) ? i : -i) ;

        Sum task = new Sum(nums, 0, nums.length);

        // Запуск ForkJoinTasks. Обратите внимание, что в данном
        // случае метод invoke() возвращает результат.
        double summation = fjp.invoke(task);

        System.out.println("Summation " + summation);
    }
}
```

Вот вывод этой программы.

```
Summation -2500.0
```

В данной программе есть несколько интересных моментов. В первую очередь обратите внимание на то, что эти две подзадачи выполняются при вызове метода `fork()`, как показано далее.

```
subTaskA.fork();
subTaskB.fork();
```

В данном случае метод `fork()` используется потому, что он запускает задачу, но не ждет ее завершения. (Таким образом, он запускает задачу асинхронно.) Результат каждой задачи получается при вызове метода `join()`, как показано далее.

```
sum = subTaskA.join() + subTaskB.join();
```

Эти операторы ожидают завершения каждой задачи. Затем все результаты суммируются и присваиваются переменной `sum`. Таким образом, суммирование всех подзадач добавляется к вычислению суммы. И наконец, метод `compute()` заканчивает работу, возвращая значение переменной `sum`, которое будет окончательным общим количеством для первого запроса.

Существуют и другие подходы асинхронного выполнения подзадач. Например, следующая последовательность использует метод `fork()` для запуска задачи `subTaskA` и метод `invoke()` — для запуска и ожидания задачи `subTaskB`.

```
subTaskA.fork();
sum = subTaskA.join() + subTaskB.invoke();
```

Еще одним вариантом может быть непосредственный вызов метода `compute()` задачи `subTaskB`, как показано далее.

```
subTaskA.fork();
sum = subTaskA.join() + subTaskB.compute();
```

Асинхронное выполнение задач

Для инициализации задачи приведенные выше программы вызвали метод `invoke()` класса `ForkJoinPool`. Этот подход является общепринятым, когда вызывающий поток должен ждать завершения задачи (что зачастую имеет место), поскольку метод `invoke()` не завершается, пока задача не закончится. Однако вы можете запустить задачу асинхронно. При этом подходе вызывающий поток продолжает выполняться. Таким образом, и вызывающий поток, и задача выполняются одновременно. Чтобы запустить задачу асинхронно, используйте метод `execute()`, который также определяет класс `ForkJoinPool`. Две его формы представлены ниже.

```
void execute(ForkJoinTask<?> задача)
void execute(Runnable задача)
```

В обеих формах выполняемую задачу определяет параметр *задача*. Обратите внимание на то, что вторая форма позволяет определить задачу типа `Runnable`, а не `ForkJoinTask`. Таким образом, это создает мост между традиционным подходом Java к многопоточности и новой инфраструктурой Fork/Join Framework. Важно помнить, что потоки, используемые классом `ForkJoinPool`, являются демоном. Таким образом, они завершаются по завершении основного потока. В результате вам, возможно, придется поддерживать основной поток, пока задачи не закончатся.

Отмена задачи

Задача может быть отменена при вызове метода `cancel()`, который определен в классе `ForkJoinTask`. Его общая форма такова.

```
boolean cancel(boolean успешнПрерыв)
```

Метод возвращает значение `true`, если задача, для которой он был вызван, успешно отменена, и значение `false` – если задача уже отменена, закончена или не может быть отменена. В настоящее время параметр *успешнПрерыв* не используется стандартной реализацией. Обычно метод `cancel()` предназначается для вызова из кода вне задачи, поскольку задача может легко отменить себя при выходе.

При вызове метода `isCancelled()` вы можете определить, была ли задача отменена, как показано далее.

```
final boolean isCancelled()
```

Метод возвращает значение `true`, если вызывающая задача была отменена до завершения, и значение `false` – в противном случае.

Определение состояния завершения задачи

Кроме только что описанного метода `isCancelled()`, класс `ForkJoinTask` включает два других метода, которые вы можете использовать для определения состояния завершения задачи. Первый из них, метод `isCompletedNormally()`, представлен ниже.

```
final boolean isCompletedNormally()
```

Метод возвращает значение `true`, если вызывающая задача закончилась нормально, т.е. не передала исключения, и не была отменена вызовом метода `cancel()`, а в противном случае – значение `false`.

Второй метод, `isCompletedAbnormally()`, представлен здесь.

```
final boolean isCompletedAbnormally()
```

Метод возвращает значение `true`, если вызывающая задача закончила работу вследствие ее отмены или передачи исключения, а в противном случае – значение `false`.

Перезапуск задачи

Обычно вы не можете перезапустить задачу. Другими словами, как только задача заканчивает выполнение, она не может быть перезапущена. Однако вы можете повторно инициализировать состояние задачи (после ее завершения), таким образом, она может быть запущена снова. Для этого используется вызов метода `reinitialize()`, как показано далее.

```
void reinitialize()
```

Этот метод сбрасывает состояние вызывающей задачи. Однако любая модификация, внесенная в любые постоянные данные, которые используются задачей, не будет отменена. Например, если задача изменяет массив, то эта модификация не будет отменена при вызове метода `reinitialize()`.

Что исследовать

Приведенное выше обсуждение затронуло основные принципы инфраструктуры Fork/Join Framework и наиболее часто используемые методы. Однако инфраструк-

тура Fork/Join Framework — это богатая среда выполнения, включающая передовые возможности, обеспечивающие дополнительный контроль над параллельностью. Хотя изучение всех проблем и нюансов параллельного программирования и инфраструктуры Fork/Join Framework выходит за рамки этой книги, некоторые из средств, предоставляемых классами `ForkJoinTask` и `ForkJoinPool`, здесь упоминаются.

Некоторые иные средства класса `ForkJoinTask`

Как уже упоминалось, такие методы, как `invokeAll()` и `fork()`, могут быть вызваны только внутри класса `ForkJoinTask`. Обычно соблюсти это правило просто, но в некоторых случаях у вас может быть код, способный выполняться как внутри, так и вне задачи. Вы можете определить, выполняется ли ваш код внутри задачи, вызвав метод `inForkJoinPool()`.

Вы можете преобразовать объект интерфейса `Runnable` или `Callable` в объект класса `ForkJoinTask` при помощи метода `adapt()`, определенного в классе `ForkJoinTask`. У этого метода есть три формы: для преобразования объекта интерфейса `Callable`; для объекта интерфейса `Runnable`, который не возвращает результат; и для объекта интерфейса `Runnable`, который действительно возвращает результат. В случае интерфейса `Callable` выполняется метод `call()`, а в случае интерфейса `Runnable` — метод `run()`.

При вызове метода `getQueuedTaskCount()` вы можете получить приблизительное количество задач, находящихся в очереди вызывающего потока. При вызове метода `getSurplusQueuedTaskCount()` можно получить приблизительное количество задач, имеющихся у вызывающего потока в очереди, которое превышает количество других потоков в пуле, который мог бы "захватить" их. Помните, захват задачи в инфраструктуре Fork/Join Framework — это единственный путь получения высокой эффективности. Хотя этот процесс является автоматическим, в некоторых случаях информация о нем может оказаться полезной при оптимизации производительности.

Класс `ForkJoinTask` определяет два метода, которые начинаются с префикса `quietly`. Они представлены ниже.

`final void quietlyJoin()`	Присоединяет задачу, но не возвращает результат и не передает исключений
`final void quietlyInvoke()`	Вызывает задачу, но не возвращает результат и не передает исключений

В основном, эти методы подобны своим упрощенным аналогам, кроме того, что они не возвращают значений и не передают исключений.

Вы можете попытаться "отменить вызов" задачи (другими словами, исключить ее из расписания) при вызове метода `tryUnfork()`.

Класс `ForkJoinTask` реализует интерфейс `Serializable`. Таким образом, он разрешает сериализацию. Однако сериализация не используется во время выполнения.

Некоторые иные средства класса `ForkJoinPool`

Одним из методов, весьма полезных при настройке приложений ветвления и объединения, является переопределенный метод `toString()` класса `ForkJoinPool`. Он отображает "дружественный" отчет о состоянии пула. Чтобы увидеть его в действии, используйте приведенную ниже последовательность для запуска и последующего ожидания задачи в классе `FJExperiment` в представленной ранее программе экспериментов с задачами.

```
// Запустить главный ForkJoinTask асинхронно.
fjp.execute(task);

// Отобразить состояние пула при ожидании.
while(!task.isDone()) {
    System.out.println(fjp);
}
```

Запустив программу вы увидите на экране серию сообщений, описывающих состояние пула. Вот один из примеров. Конечно, ваш вывод может быть другим, из-за различия в количестве процессоров, пороговых значений, нагрузки задач и т.д.

```
java.util.concurrent.ForkJoinPool@141d683[Running, parallelism = 2, size
= 2, active = 0, running = 2, steals = 0, tasks = 0, submissions = 1]
```

Вы можете определить, не бездействует ли пул в настоящее время при вызове метода `isQuiescent()`. Он возвращает значение `true`, если у пула нет никаких активных потоков, и значение `false` — в противном случае.

Вы можете получить количество рабочих потоков, находящихся в пуле в настоящее время, при вызове метода `getPoolSize()`. При вызове метода `getActiveThreadCount()` можно получить приблизительное количество активных потоков в пуле.

Чтобы завершить работу пула, вызовите метод `shutdown()`. Текущие задачи все еще будут выполняться, но никаких новых задач запущено быть не может. Чтобы остановить пул немедленно, вызовите метод `shutdownNow()`. В данном случае делается попытка отменить текущие задачи. При вызове метода `isShutdown()` вы можете определить, закрывается ли пул. Он возвращает значение `true`, если пул был закрыт, и значение `false` — в противном случае. Чтобы определить, был ли пул закрыт и все ли задачи закончены, вызовите метод `isTerminated()`.

Некоторые советы относительно ветвления/объединения

Вот несколько советов, способных помочь вам избежать некоторых из наиболее неприятных проблем, связанных с использованием инфраструктуры Fork/Join Framework. Во-первых, избегайте использования слишком низкого порогового значения последовательного выполнения. Обычно ошибки допущение на высоком уровне лучше, чем ошибки допущение на низком. Если пороговое значение слишком низко, на создание и переключение задач может уйти времени больше, чем на их обработку. Во-вторых, обычно лучше использовать уровень параллелизма, заданный по умолчанию. Если вы определяете меньшее значение, это может значительно уменьшить преимущество использования инфраструктуры Fork/Join Framework.

Обычно класс `ForkJoinTask` не должен использовать синхронизированные методы или блоки кода. Кроме того, вы обычно не будете использовать метод `compute()` с другими типами синхронизации, такими как семафор. (Однако новый класс `Phaser` может быть использован, поскольку он совместим с механизмом ветвления/объединения.) Помните, что основная идея класса `ForkJoinTask` состоит в стратегии "разделяй и властвуй". Такой подход обычно не применяется в ситуациях, где необходима внешняя синхронизация. Кроме того, избегайте ситуаций, в которых ввод-вывод способен вызвать существенную блокировку. Поэтому класс `ForkJoinTask` обычно не будет обслуживать ввод-вывод. Проще говоря, чтобы лучше использовать инфраструктуру Fork/Join Framework, задача должна выполнять вычисление, которое может осуществляться без внешней блокировки или синхронизации.

Последний момент: за исключением необычных обстоятельств, не делайте никаких предположений о среде выполнения, в которой будет работать ваш код. Это значит, что вы не должны подразумевать, что будет доступно некое количество процессоров или что на характеристики выполнения вашей программы не будут влиять другие процессы, выполняющиеся в то же время.

Параллельные утилиты в сравнении с традиционным подходом в Java

Если принять во внимание ту мощность и гибкость, которую предлагают новые параллельные утилиты, то возникает следующий резонный вопрос: могут ли они служить заменой традиционному для Java подходу к реализации многопоточности и синхронизации? Однозначно нет! Первоначальная поддержка многопоточности и встроенные функциональные возможности синхронизации по-прежнему следует реализовать во множестве программ Java, аплетах и сервлетах. Например, методы `synchronized`, `wait()` и `notify()` предлагают элегантные решения широкого круга задач. Однако если вам понадобится расширить возможности управления, то для этих целей можно воспользоваться параллельными утилитами. Кроме того, новая инфраструктура Fork/Join Framework предлагает мощнейший способ интеграции технологий параллельного программирования в ваши более сложные приложения.

ГЛАВА 28

Регулярные выражения и другие пакеты

Изначально язык Java включал восемь пакетов, называемых *API ядра* (core API). С каждым последующим выпуском API пополнялся новыми пакетами. На сегодняшний день API Java содержит большое количество пакетов. Многие из них являются специализированными и не рассматриваются в этой книге. Тем не менее четыре пакета мы все же рассмотрим — `java.util.regex`, `java.lang.reflect`, `java.rmi` и `java.text`. Эти пакеты поддерживают обработку регулярных выражений, рефлексию, дистанционный вызов методов и обработку текстов.

Пакет *регулярных выражений* позволит выполнять сложные операции сопоставления с шаблонами. В этой главе дан углубленный анализ этого пакета и предложены многочисленные его примеры. *Рефлексией* называется способность программного обеспечения к самоанализу.

Рефлексия является неотъемлемой частью технологии Java Bean, которая рассматривается в главе 29. *Дистанционный вызов методов* (Remote Method Invocation — RMI) позволяет создавать приложения Java, которые будут распределяться среди нескольких компьютеров. В этой главе будет предложен один простой клиент-серверный пример, в котором используется RMI. Возможности *форматирования текста*, предлагаемые пакетом `java.text`, имеют самое широкое применение. В этой главе будут рассмотрены примеры форматирования строк даты и времени.

Пакеты API ядра

В табл. 28.1 приведен список всех пакетов API ядра (в пространстве имен `java`) с указанием их назначения.

Таблица 28.1. Пакеты API ядра

Пакет	Основное назначение
`java.applet`	Поддерживает процесс создания аплетов
`java.awt`	Предоставляет возможности графических пользовательских интерфейсов
`java.awt.color`	Поддерживает цветовые пространства и профили
`java.awt.datatransfer`	Передает данные в системный буфер обмена и обратно
`java.awt.dnd`	Поддерживает операции перетаскивания
`java.awt.event`	Обрабатывает события
`java.awt.font`	Представляет различные типы шрифтов
`java.awt.geom`	Позволяет работать с геометрическими формами

Продолжение табл. 28.1

Пакет	Основное назначение
java.awt.im	Позволяет вводить японские, китайские и корейские символы в компоненты редактирования текста
java.awt.im.spi	Поддерживает альтернативные устройства ввода
java.awt.image	Обрабатывает изображения
java.awt.image.renderable	Поддерживает изображения с независимой визуализацией
java.awt.print	Поддерживает общие свойства печати
java.beans	Позволяет создавать компоненты программного обеспечения
java.beans.beancontext	Обеспечивает среду выполнения для компонентов Java Bean
java.io	Вводит и выводит данные
java.lang	Обеспечивает базовые функциональные возможности
java.lang.annotation	Поддерживает аннотации (метаданные)
java.lang.instrument	Поддерживает инструментальные средства для программ
java.lang.invoke	Поддерживает динамические языки
java.lang.management	Поддерживает управление средой выполнения
java.lang.ref	Позволяет взаимодействовать со сборщиком "мусора"
java.lang.reflect	Анализирует код во время выполнения
java.math	Обрабатывает большие целые и десятичные числа
java.net	Поддерживает работу в сети
java.nio	Пакет верхнего уровня для классов NIO. Инкапсулирует буферы
java.nio.channels	Инкапсулирует каналы, используемые в системе NIO
java.nio.channels.spi	Поддерживает провайдеры служб для каналов
java.nio.charset	Инкапсулирует наборы символов
java.nio.charset.spi	Поддерживает провайдеры служб для наборов символов
java.nio.file	Обеспечивает поддержку NIO для файлов
java.nio.file.attribute	Поддерживает файловые атрибуты NIO
java.nio.file.spi	Поддерживает провайдеры служб NIO для файлов
java.rmi	Обеспечивает дистанционный вызов методов
java.rmi.activation	Активизирует постоянные объекты
java.rmi.dgc	Управляет распределенным сбором "мусора"
java.rmi.registry	Отображает имена как ссылки на дистанционные объекты
java.rmi.server	Поддерживает дистанционный вызов методов
java.security	Обрабатывает сертификаты, ключи, дайджесты, подписи и другие функции, связанные с безопасностью
java.security.acl	Поддерживает списки управления доступом
java.security.cert	Выполняет синтаксический анализ и управление сертификатами
java.security.interfaces	Определяет интерфейсы для ключей DSA (Digital Signature Algorithm – алгоритм цифровой подписи)

Пакет	Основное назначение
java.security.spec	Определяет параметры ключей и алгоритмов
java.sql	Взаимодействует с базой данных SQL (Structured Query Language – язык структурированных запросов)
java.text	Форматирует, производит поиск и манипулирует текстом
java.text.spi	Поддерживает провайдеры служб для классов форматирования текста в Java.text
java.util	Содержит общие утилиты
java.util.concurrent	Поддерживает параллельные утилиты
java.util.concurrent.atomic	Поддерживает атомарные (т.е. неделимые) операции в отношении переменных без использования блокировок
java.util.concurrent.locks	Поддерживает синхронизационные блокировки
java.util.jar	Создает и считывает файлы JAR
java.util.logging	Поддерживает регистрацию информации, связанной с выполнением программы
java.util.prefs	Инкапсулирует информацию, связанную с предпочтениями пользователя
java.util.regex	Поддерживает обработку регулярных выражений
Java.util.spl	Поддерживает провайдеры служб вспомогательных классов в пакете java.util
java.util.zip	Считывает и записывает упакованные и распакованные файлы ZIP

Обработка регулярных выражений

Пакет java.util.regex поддерживает обработку регулярных выражений. Используемый здесь термин *регулярное выражение* (regular expression) относится к строке символов, описывающей последовательность символов. Это общее описание, называемое *шаблоном*, впоследствии может использоваться для поиска совпадений в других последовательностях символов. Регулярные выражения могут определять групповые символы, наборы символов и разнообразные квантификаторы. Таким образом, вы можете задать регулярное выражение, представляющее общую форму, которая может совпадать с несколькими различными специфическими последовательностями символов.

Существует два класса, поддерживающих обработку регулярных выражений, – Pattern и Matcher. Эти классы работают вместе. Класс Pattern применяется для задания регулярного выражения. Сопоставление шаблона с последовательностью символов реализуется с помощью класса Matcher.

Класс Pattern

В классе Pattern нет определений конструкторов. Наоборот, шаблон формируется с помощью вызова фабричного метода compile(). Одна из форм этого метода показана далее.

```
static Pattern compile(String шаблон)
```

Здесь *шаблон* представляет регулярное выражение, которое вы хотите использовать. Метод `compile()` преобразует строку *шаблон* в шаблон, который можно использовать для сопоставления, осуществляемого с помощью класса `Matcher`. Этот метод возвращает объект класса `Pattern`, содержащий данный шаблон.

После того как создадите объект класса `Pattern`, будете использовать его для создания объекта класса `Matcher`. Для этого вызывается фабричный метод `matcher()`, определяемый классом `Pattern`.

```
Matcher matcher(CharSequence строка)
```

Здесь *строка* — это последовательность символов, которая будет сопоставляться с шаблоном. Она называется *входной последовательностью*. Интерфейс `CharSequence` определяет набор символов только для чтения. Помимо всего прочего, он реализуется классом `String`. Таким образом, вы передаете строку методу `matcher()`.

Класс `Matcher`

Этот класс не имеет конструкторов. Наоборот, вы создаете объект класса `Matcher` при помощи вызова фабричного метода `matcher()`, определяемого в классе `Pattern`, о чем мы уже говорили. После того как создадите объект класса `Matcher`, вы будете использовать его методы для выполнения различных операций по сопоставлению с шаблонами.

Самым простым методом сопоставления с шаблоном является метод `matches()`, который просто определяет, совпадает ли последовательность символов с шаблоном. Этот метод показан ниже.

```
boolean matches()
```

Он возвращает значение `true`, если последовательность и шаблон совпадают, в противном случае он возвращает значение `false`. Следует иметь в виду, что с шаблоном должна совпадать не вся последовательность, а только ее часть (подпоследовательность).

Чтобы определить, совпадает ли подпоследовательность с входящей последовательностью, используется метод `find()`. Ниже показана одна из его версий.

```
boolean find()
```

Этот метод возвращает значение `true`, если имеет место совпадение, в противном случае он возвращает значение `false`. Метод можно вызывать неоднократно и находить все совпадающие подпоследовательности. При каждом вызове метода `find()` сравнение начинается с того места, где было закончено предыдущее.

Строку, содержащую последнюю совпавшую последовательность, можно получить с помощью метода `group()`. Одна из его форм показана ниже.

```
String group()
```

Метод возвращает совпавшую строку. Если совпадение не было обнаружено, передается исключение `IllegalStateException`.

С помощью метода `start()` можно получить индекс внутри входной последовательности текущего совпадения. Индекс, следующий за окончанием текущего совпадения, можно получить с помощью метода `end()`. Эти методы показаны ниже.

```
int start()
int end()
```

Каждый из них в случае отсутствия совпадения передает исключение `IllegalStateException`.

Каждую совпавшую последовательность можно заменить другой последовательностью, если вызвать метод `replaceAll()`.

```
String replaceAll(String новСтр)
```

Здесь параметр *новСтр* определяет новую последовательность символов, которая заменит последовательности, совпавшие с шаблоном. Обновленная входная последовательность будет возвращена в качестве строки.

Синтаксис регулярного выражения

Прежде чем продемонстрировать работу классов `Pattern` и `Matcher`, необходимо объяснить, как формируется регулярное выражение. Хотя ни одно из правил не является сложным, их существует большое количество, и полностью описать правила в этой книге невозможно. Однако здесь вы найдете несколько наиболее распространенных конструкций.

В общем случае регулярное выражение состоит из обычных символов, классов символов (наборов символов), групповых символов и квантификаторов. Обычный символ сопоставляется по принципу "как есть". Таким образом, если шаблон содержит пару символов `"xy"`, то единственной входящей последовательностью, которая может совпасть с этим шаблоном, является `"xy"`. Символы вроде новой строки и табуляции определяются при помощи стандартных управляющих последовательностей, которые начинаются со слеша (`\`). Например, символ новой строки определяется последовательностью `\n`. В языке регулярных выражений обычный символ называется также литералом.

Класс символов является набором символов. Класс символов можно задать, заключив символы класса в квадратные скобки. Например, класс `[wxyz]` совпадает с w, x, y или z. Чтобы задать обратный (инвертированный) набор, перед символами необходимо поставить знак `^`. Например, класс `[^wxyz]` совпадает с любым символом, за исключением w, x, y или z. С помощью дефиса указывается диапазон символов. Например, чтобы задать класс символов, которые будут совпадать с цифрами от 1 до 9, используется запись `[1-9]`.

Групповым символом является символ точки (`.`), который совпадает с любым символом вообще. Таким образом, шаблон, состоящий из `"."`, будет совпадать с любой из следующих (и любыми другими) входящих последовательностей: `"A"`, `"a"`, `"x"` и т.д.

Квантификатор определяет, сколько раз совпадает выражение. В табл. 28.2 представлены возможные квантификаторы.

Таблица 28.2. Квантификаторы, применяемые в регулярных выражениях

Квантификатор	Описание
`+`	Совпадает один или более раз
`*`	Совпадает нуль или более раз
`?`	Совпадает нуль или один раз

Например, шаблон `"x+"` будет совпадать с `"x"`, `"xx"`, `"xxx"` и т.п.

И еще одно: вообще, если вы определите неверное выражение, будет создано исключение `PatternSyntaxException`.

Пример совпадения с шаблоном

Чтобы лучше понять, как осуществляется сопоставление с шаблоном в регулярном выражении, давайте рассмотрим несколько примеров. В первом из них, показанном далее, производится поиск совпадения с литеральным шаблоном.

```
// Пример простого 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(); // поиск совпадения

        System.out.println("Проверка совпадения Java с Java.");
        if(found) System.out.println("Совпадает");
        else System.out.println("Не совпадает");

        System.out.println();

        System.out.println("Проверка совпадения Java с Java 7.");
        mat = pat.matcher("Java 7"); // создание нового обнаружителя
                                     // совпадений

        found = mat.matches(); // поиск совпадения

        if(found) System.out.println("Совпадает");
        else System.out.println("Не совпадает");
    }
}
```

Ниже представлены результаты выполнения этой программы.

```
Проверка совпадения Java с Java.
Совпадает
Проверка совпадения Java с Java 7.
Не совпадает
```

Давайте проанализируем эту программу. Она начинается с создания шаблона, состоящего из последовательности `"Java"`. После этого для данного шаблона создается объект класса `Matcher` с входящей последовательностью `"Java"`. Затем вызывается метод `matches()`, с помощью которого определяется, совпадает ли входящая последовательность с шаблоном. Так как последовательность и шаблон одинаковые, метод `matches()` возвращает значение `true`. Затем создается новый объект класса `Matcher` с входящей последовательностью `"Java 7"` и опять вызывается метод `matches()`. В этом случае шаблон и входящая последовательность отличаются друг от друга, поэтому совпадение не обнаруживается. Помните, что метод `matches()` возвращает значение `true` только в том случае, когда входящая последовательность в точности совпадает с шаблоном. Он не возвращает значение `true` просто потому, что подпоследовательность не совпадает с шаблоном.

Чтобы определить, содержит ли входная последовательность подпоследовательность, совпадающую с шаблоном, можно вызвать метод `find()`. Рассмотрим следующую программу.

```
// Использование метода find() для нахождения подпоследовательности.
import java.util.regex.*;

class RegExpr2 {
    public static void main(String args[]) {
        Pattern pat = Pattern.compile("Java");
        Matcher mat = pat.matcher("Java 7");
```

```
        System.out.println("Поиск Java в Java 7.");

        if(mat.find()) System.out.println("Подпоследовательность
                                          найдена");
        else System.out.println("Совпадений нет");
    }
}
```

Ниже показаны результаты выполнения этой программы.

```
Поиск Java в Java 7.
Подпоследовательность найдена
```

В данном случае метод `find()` ищет подпоследовательность `"Java"`.

Метод `find()` можно использовать для поиска во входящей последовательности повторяющихся совпадений с шаблоном, поскольку каждый вызов метода `find()` начинается с того места, где был завершен предыдущий. Например, следующая программа находит два случая совпадения с шаблоном `"test"`.

```
// Использование метода find() для нахождения нескольких
// подпоследовательностей.
import java.util.regex.*;

class RegExpr3 {
    public static void main(String args[]) {
        Pattern pat = Pattern.compile("test");

        Matcher mat = pat.matcher("test 1 2 3 test");
        while(mat.find()) {
            System.out.println("test найдена по индексу " + mat.start());
        }
    }
}
```

Ниже представлены результаты выполнения этой программы.

```
test найдена по индексу 0
test найдена по индексу 11
```

Как можно видеть из этих результатов, было найдено два случая совпадения. Программа использует метод `start()` для получения индекса каждого совпадения.

Использование групповых символов и квантификаторов

В предыдущей программе была показана общая технология использования классов `Pattern` и `Matcher`, но она не раскрывает полностью их возможности. Реальное преимущество обработки регулярных выражений оценить невозможно, не используя групповые символы и квантификаторы. Для начала рассмотрим следующий пример, в котором квантификатор + используется для сопоставления любой произвольной последовательности 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());
```

```
    }
}
```

Ниже представлены результаты выполнения этой программы.

```
Совпадение: W
Совпадение: WW
Совпадение: WWW
```

Как можно видеть из результатов, комбинация `"W+"` в регулярном выражении совпадает с последовательностью символов W какой угодно длины.

В следующей программе групповой символ используется для формирования шаблона, который будет сопоставляться с любой последовательностью, начинающейся с `e` и заканчивающейся `d`. Для этого в ней используется групповой символ точки и квантификатор +.

```
// 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());
    }
}
```

Результаты выполнения этой программы могут быть для вас неожиданными.

```
Совпадение: extend cup end
```

Был найден только один случай совпадения — это самая длинная последовательность, которая начинается с `e` и заканчивается `d`. Вы могли предположить, что в результате выполнения будет найдено два случая совпадения: `"extend"` и `"end"`. Более длинная последовательность была обнаружена по той причине, что по умолчанию метод `find()` осуществляет сопоставление с самой длинной последовательностью, которая соответствует всему шаблону. Это называется *поглощающим поведением*. Можно задать *принудительное поведение*, если добавить в комбинацию квантификатор `?`, как показано в следующем варианте программы. В результате будет получен более короткий шаблон для сопоставления.

```
// Использование квантификатора ?.
import java.util.regex.*;

class RegExpr6 {
    public static void main(String args[]) {
        // Использование поведения принудительного сопоставления.
        Pattern pat = Pattern.compile("e.+?d");
        Matcher mat = pat.matcher("extend cup end table");

        while(mat.find())
            System.out.println("Совпадение: " + mat.group());
    }
}
```

Ниже представлены результаты выполнения этой программы.

```
Совпадение: extend
Совпадение: end
```

Как можно заметить из результатов, шаблон "e.+?d" будет совпадать с более короткой последовательностью, которая начинается с e и заканчивается d. Таким образом, будет обнаружено два случая совпадения.

Работа с классами символов

Иногда возникает необходимость в сопоставлении в любом порядке состоящей из одного или нескольких символов какой-нибудь последовательности, которая будет частью набора символов. Например, чтобы сопоставить целые слова, вам потребуется сопоставить любую последовательность букв алфавита. Проще всего это сделать с использованием класса символов, определяющего набор символов. Напомним, чтобы сформировать класс символов, необходимых для сопоставления, их потребуется заключить в квадратные скобки. Например, чтобы сопоставить символы нижнего регистра от a до z, используется запись [a-z]. В следующей программе демонстрируется этот способ.

```
// Использование класса символов.
import java.util.regex.*;

class RegExpr7 {
    public static void main(String args[]) {
        // Сопоставление слов 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 показаны результаты выполнения этой программы.

```
Совпадение: this
Совпадение: is
Совпадение: a
Совпадение: test
```

Использование метода replaceAll()

С помощью метода replaceAll(), определенного в классе Matcher, можно выполнять полноценные операции поиска и замены, в которых используются регулярные выражения. Например, в следующей программе каждая последовательность "Jon" заменяется последовательностью "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("Начальная последовательность: " + str);
        str = mat.replaceAll("Eric ");

        System.out.println("Измененная последовательность: " + str);
    }
}
```

Ниже представлены результаты выполнения этой программы.

```
Начальная последовательность: Jon Jonathan Frank Ken Todd
Измененная последовательность: Eric Eric Frank Ken Todd
```

Поскольку регулярное выражение `"Jon.*?"` сопоставляет любую строку, начинающуюся с последовательности `Jon`, за которой больше нет символов или стоит несколько символов, заканчивающихся пробелом, его можно применить для сопоставления и замены имен Jon и Jonathan именем Eric. Такую замену нельзя было бы выполнить, если бы не было возможности осуществить сопоставление с шаблоном.

Использование метода `split()`

С помощью метода `split()` можно сократить входящую последовательность до ее индивидуальных лексем, разделяемых пробелами, запятыми, точками и знаками восклицания.

```
// Использование метода split().
import java.util.regex.*;

class RegExpr9 {
    public static void main(String args[]) {

        // Сопоставление слов 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]);
    }
}
```

Ниже представлены результаты выполнения программы.

```
Следующая лексема: one
Следующая лексема: two
Следующая лексема: alpha9
Следующая лексема: 12
Следующая лексема: done
```

Как можно заметить из результатов, входная последовательность сокращается до ее индивидуальных лексем. Обратите внимание: разделители не включаются.

Два варианта сопоставления с шаблоном

Хотя описанные методы сопоставления с шаблонами характеризуются высокой степенью гибкости и хорошей производительностью, существует два варианта, применение которых в некоторых обстоятельствах может быть очень полезным. Если вам необходимо только одноразовое сопоставление с шаблоном, можете использовать метод `matches()`, определяемый классом `Pattern`.

```
static boolean matches(String шаблон, CharSequence строка)
```

Метод возвращает значение `true`, если шаблон *`шаблон`* совпадает со строкой в параметре *`строка`*, в противном случае он возвращает значение `false`. Этот метод автоматически компилирует *`шаблон`*, после чего производит поиск совпадения. Если использовать один и тот же шаблон несколько раз подряд, то применение метода `matches()` будет менее эффективным, чем компиляция шаблона и использование методов сопоставления с шаблонами, определяемых классом `Matches`, о чем было сказано ранее.

Сопоставление с шаблоном можно также выполнять с помощью метода `matches()`, реализуемого классом `String`. Этот метод показан ниже.

```
boolean matches(String шаблон)
```

Если вызывающая строка совпадает с регулярным выражением в параметре *шаблон*, то метод `matches()` возвращает значение `true`. В противном случае он возвращает значение `false`.

Изучение регулярных выражений

Обзор регулярных выражений, предложенный в этом разделе, лишь отчасти раскрывает их настоящие возможности. Поскольку синтаксический анализ текста, манипулирование и разбиение на лексемы являются частью программирования, то вы, скорее всего, придете к выводу, что подсистема регулярных выражений в Java является хорошим инструментом, который можно использовать с большой пользой. Поэтому вам следует потратить некоторое время на изучение свойств регулярных выражений. Поэкспериментируйте с несколькими различными типами шаблонов и входящих последовательностей. После того как разберетесь с тем, как осуществляется сопоставление с шаблонами в регулярных выражениях, вы поймете, что использование этого подхода будет очень полезным при решении многих задач программирования.

Рефлексия

Рефлексия (reflection) – это способность программного обеспечения к самоанализу. Эта способность обеспечивается пакетом `java.lang.reflect` и элементами класса `Class`. Рефлексия является важным свойством, особенно для Java Beans. С ее помощью вы сможете анализировать компоненты программного обеспечения и описывать их свойства динамически во время выполнения, а не компиляции. Например, с помощью рефлексии можно определить, какие методы, конструкторы и поля поддерживает конкретный класс. О рефлексии мы начали говорить в главе 12. В настоящей главе продолжим эту тему.

Пакет `java.lang.reflect` имеет несколько интерфейсов. Особый интерес представляет интерфейс `Member`, определяющий методы, которые позволяют получить информацию о поле, конструкторе или методе класса. В этом пакете существует также еще восемь классов. Все они перечислены в табл. 28.3.

Таблица 28.3. Классы, определенные в пакете `java.lang.reflect`

Класс	Основное назначение
`AccessibleObject`	Позволяет обходить стандартные проверки управления доступом
`Array`	Позволяет динамически создавать массивы и манипулировать ими
`Constructor`	Предлагает информацию о конструкторе
`Field`	Предлагает информацию о поле
`Method`	Предлагает информацию о методе
`Modifier`	Предлагает информацию о модификаторах доступа к классу и его членам
`Proxy`	Поддерживает динамические прокси-классы
`ReflectPermission`	Разрешает рефлексию закрытых или защищенных членов класса

Следующее приложение иллюстрирует простой вариант использования свойств рефлексии в Java. Оно выводит на экран конструкторы, поля и методы класса `java.awt.Dimension`. Программа начинается с использования метода `forName()` класса `Class` для получения объекта класса `java.awt.Dimension`. После того как он будет получен, используются методы `getConstructors()`, `getFileds()` и `getMethods()` для анализа этого объекта класса. Они возвращают массивы класса `Constructor`, `Field` и `Method`, содержащие информацию об объекте.

Классы `Constructor`, `Field` и `Method` определяют несколько методов, которые могут использоваться для получения информации об объекте. Вам придется изучить их самостоятельно. Однако каждый из них поддерживает метод `toString()`. Таким образом, использование объектов класса `Constructor`, `Field` и `Method` в качестве параметров метода `println()` не представляет сложности, что можно видеть из самой программы.

```
// 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("Исключение: " + e);
        }
    }
}
```

Ниже приводятся результаты выполнения этой программы (у вас они могут немного 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 следующем примере используются особенности рефлексии в Java для получения открытых методов класса. Программа начинается с реализации класса `A`. Метод `getClass()` применяется к этой ссылке на объект и возвращает объект класса `Class` для класса `A`. Метод `getDeclaredMethods()` возвращает массив объектов класса `Method`, который описывает только методы, объявленные в этом классе. Методы, унаследованные от суперклассов, например класса `Object`, не включаются.

После этого обрабатывается каждый элемент массива `methods`. Метод `getModifiers()` возвращает значение типа `int`, содержащее флаги, которые описывают, какие модификаторы доступа применимы к этому элементу. Класс `Modifier` предлагает набор методов `isX`, перечисленных в табл. 28.4, которые можно использовать для проверки этого значения. Например, статический метод `isPublic()` возвращает значение `true`, если параметр включает модификатор доступа `public`. В противном случае он возвращает значение `false`.

Таблица 28.4. Методы класса `Modifier`, определяющие модификаторы доступа

Методы	Описание
`static boolean isAbstract(int` *значение*`)`	Возвращает значение `true`, если *значение* имеет установленный флаг `abstract`, в противном случае возвращает значение `false`
`static boolean isFinal(int` *значение*`)`	Возвращает значение `true`, если *значение* имеет установленный флаг `final`, в противном случае возвращает значение `false`
`static boolean isInterface(int` *значение*`)`	Возвращает значение `true`, если *значение* имеет установленный флаг `interface`, в противном случае возвращает значение `false`
`static boolean isNative(int` *значение*`)`	Возвращает значение `true`, если *значение* имеет установленный флаг `native`, в противном случае возвращает значение `false`
`static boolean isPrivate(int` *значение*`)`	Возвращает значение `true`, если *значение* имеет установленный флаг `private`, в противном случае возвращает значение `false`
`static boolean isProtected(int` *значение*`)`	Возвращает значение `true`, если *значение* имеет установленный флаг `protected`, в противном случае возвращает значение `false`
`static boolean isPublic(int` *значение*`)`	Возвращает значение `true`, если *значение* имеет установленный флаг `public`, в противном случае возвращает значение `false`

Окончание табл. 28.4

Методы	Описание
`static boolean isStatic(int` *значение*`)`	Возвращает значение `true`, если *значение* имеет установленный флаг `static`, в противном случае возвращает значение `false`
`static boolean isStrict(int` *значение*`)`	Возвращает значение `true`, если *значение* имеет установленный флаг `strict`, в противном случае возвращает значение `false`
`static boolean isSynchronized(int` *значение*`)`	Возвращает значение `true`, если *значение* имеет установленный флаг `synchronized`, в противном случае возвращает значение `false`
`static boolean isTransient(int` *значение*`)`	Возвращает значение `true`, если *значение* имеет установленный флаг `transient`, в противном случае возвращает значение `false`
`static boolean isVolatile(int` *значение*`)`	Возвращает значение `true`, если *значение* имеет установленный флаг `volatile`, в противном случае возвращает значение `false`

В следующей программе, если метод является открытым, с помощью метода `getName()` будет получено его имя и выведено на экран.

```
// 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("Исключение: " + e);
        }
    }
}

class A {
    public void a1() {
    }
    public void a2() {
    }
    protected void a3() {
    }
    private void a4() {
    }
}
```

Ниже приводятся результаты выполнения этой программы.

```
Открытые методы:
a1
a2
```

Начиная с JDK 7 класс Modifier содержит также ряд статических методов, которые возвращают тип модификаторов, которые могут быть применены к определенному типу программного элемента. Вот эти методы.

```
static int classModifiers()
static int constructorModifiers()
static int fieldModifiers()
static int interfaceModifiers()
static int methodModifiers()
```

Например, метод methodModifiers() возвращает модификаторы, которые могут быть применены к методу. Каждый метод возвращает упакованные в переменную типа int флаги, которые указывают допустимые модификаторы. Значения модификаторов определяются константами PROTECTED, PUBLIC, PRIVATE, STATIC, FINAL и так далее, определенными в классе Modifier.

Дистанционный вызов методов

Дистанционный вызов методов (Remote Method Invocation – RMI) позволяет объектам Java, выполняющимся на одном компьютере, вызывать методы объектов Java, которые выполняются на другом компьютере. Это важная возможность, поскольку она позволяет создавать распределенные приложения. Несмотря на то что обсуждение RMI выходит за рамки этой книги, в приведенном ниже упрощенном примере демонстрируются основные принципы этого подхода.

Клиент-серверное приложение, использующее RMI

В этом разделе предложено поэтапное руководство для создания простого клиент-серверного приложения с использованием RMI. Сервер получает запрос от клиента, обрабатывает его и возвращает результат. В этом примере запрос состоит из двух чисел. Сервер суммирует их и возвращает полученный результат.

Этап первый: ввод и компиляция исходного кода

Это приложение использует четыре исходных файла. В первом файле, AddServerIntf.java, определяется дистанционный интерфейс, который будет предоставлен сервером. Он содержит один метод, принимающий два параметра double, и возвращает их сумму. Все дистанционные интерфейсы должны расширять интерфейс Remote, который является частью пакета java.rmi. Интерфейс Remote не определяет членов. Он предназначен лишь для того, чтобы показать, что интерфейс использует дистанционные методы. Все дистанционные методы могут передать исключение RemoteException.

```
import java.rmi.*;
public interface AddServerIntf extends Remote {
    double add(double d1, double d2) throws RemoteException;
}
```

Во втором исходном файле, AddServerImpl.java, реализуется дистанционный интерфейс. Реализовать метод add() несложно. Дистанционные объекты обычно расширяют UnicastRemoteObject, который обеспечивает функциональные возможности, необходимые для того, чтобы сделать доступными объекты для дистанционных компьютеров.

```
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 исходный файл, `AddServer.java`, содержит главную программу для компьютера-сервера. Его главная задача – обновлять реестр RMI на этом компьютере. Это делается с помощью метода `rebind()` класса `Naming` (пакет `java.rmi`). Этот метод связывает имя со ссылкой на объект. Первым параметром метода `rebind()` является строка, которая присваивает серверу имя `"AddServer"`. Его второй параметр является ссылкой на экземпляр класса `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("Исключение: " + e);
        }
    }
}
```

В четвертом исходном файле, `AddClient.java`, реализуется клиентская сторона распределенного приложения. Этому файлу требуется три аргумента командной строки. Первый из них является IP-адресом или именем компьютера-сервера. Второй и третий параметры – два числа, которые необходимо суммировать.

Приложение начинается с формирования строки, имеющей синтаксическую структуру адреса URL. Этот адрес URL использует протокол `rmi`. Строка включает IP-адрес или имя сервера и строку `"AddServer"`. Затем программа вызывает метод `lookup()` класса `Naming`. Этот метод принимает один параметр, адрес URL `rmi`, и возвращает ссылку на объект типа `AddServerIntf`. Впоследствии всякий дистанционный вызов метода можно будет направлять этому объекту.

Затем программа отображает параметры и вызывает дистанционный метод `add()`, который возвращает результат суммирования, выводимый на экран.

```
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("Исключение: " + e);
        }
    }
}
```

После того как введете весь код, используйте `javac` для компиляции четырех созданных вами исходных файлов.

Этап второй: создание заглушки вручную, если нужно

Прежде чем сможете работать с клиентом и сервером, нужно создать заглушку. В контексте RMI *заглушка* представляет собой объект Java, который находится на компьютере-клиенте. Ее функция заключается в представлении тех же интерфейсов, которые предлагает дистанционный сервер. Дистанционные вызовы методов, создаваемые клиентом, направляются в заглушку. Заглушка работает с остальными частями системы RMI для формирования запроса, отправляемого дистанционному компьютеру.

Дистанционный метод может принимать параметры, которые представляют собой простые типы или объекты. В последнем случае объект может иметь ссылки на другие объекты. Вся эта информация должна быть отправлена дистанционному компьютеру. То есть объект, передаваемый в качестве аргумента вызову дистанционного метода, должен быть сериализирован и отправлен на дистанционный компьютер. Вспомните, в главе 19 мы говорили, что при сериализации также рекурсивно обрабатываются все объекты, на которые имеются ссылки.

Если клиенту необходимо вернуть ответ, то весь процесс происходит в обратном порядке. Обратите внимание на то, что сериализация и десериализация используются также и при возвращении объектов клиенту.

До Java 5 заглушки приходилось создавать вручную, при помощи компилятора `rmic`. Для современных версий Java этот этап необязателен. Но если вы работаете в устаревшей системе, то для создания заглушки можете использовать компилятор `rmic` следующим образом.

```
rmic AddServerImpl
```

Эта команда создает файл `AddServerImpl_Stub.class`. В случае использования `rmic` убедитесь в том, что текущий каталог включен в переменную `CLASSPATH`.

Этап третий: установка файлов на компьютерах сервера и клиента

Файлы `AddClient.class`, `AddServerImpl_Stub.class` (если нужно) и `AddServerIntf.class` скопируйте в каталог на компьютере-клиенте, а файлы `AddServerIntf.class`, `AddServerImpl.class`, `AddServerImpl_Stub.class` (если нужно) и `AddServer.class` — в каталог на компьютере-сервере.

На заметку! RMI предоставляет технологии динамической загрузки классов, однако они не применяются в приведенных здесь примерах. Наоборот, все файлы, используемые приложениями клиента и сервера, должны быть установлены вручную на соответствующих компьютерах.

Этап четвертый: запуск реестра RMI на компьютере-сервере

В комплект JDK входит программа `rmiregistry`, которая выполняется на стороне сервера. Она преобразует имена в ссылки на объекты. Для начала нужно проверить, включен ли каталог, в котором находятся ваши файлы, в переменную окружения `CLASSPATH`. Затем потребуется запустить RMI Registry из командной строки, как показано ниже.

```
start rmiregistry
```

После выполнения этой команды на экране появится новое окно. Это окно нужно оставлять открытым все время, пока вы будете экспериментировать с примером RMI.

Этап пятый: запуск сервера

Код сервера запускается из командной строки, как показано ниже.

```
java AddServer
```

Вспомните, что код класса `AddServer` реализует класс `AddServerImpl` и регистрирует этот объект под именем `"AddServer"`.

Этап шестой: запуск клиента

Программное обеспечение `AddClient` требует для своей работы три аргумента — имя или IP-адрес компьютера-сервера и два числа, которые будут просуммированы. Вы можете вызвать его из командной строки, используя один из следующих форматов.

```
java AddClient server1 8 9
java AddClient 11.12.13.14 8 9
```

В первой строке указывается имя сервера. Во второй строке используется его IP-адрес (11.12.13.14).

Можно попытаться выполнить этот пример и без дистанционного сервера. Для этого достаточно установить все программы на одном и том же компьютере, запустить `rmiregistry`, `AddServer` и после этого запустить `AddClient` следующим образом.

```
java AddClient 127.0.0.1 8 9
```

Здесь адрес 127.0.0.1 является адресом "ответа" для локального компьютера. Использование этого адреса позволит выполнить весь механизм RMI, не устанавливая сервер на дистанционном компьютере.

В любом случае результат выполнения этой программы будет таким.

```
Первое число: 8
Второе число: 9
Сумма: 17.0
```

На заметку! При работе с RMI в реальном приложении, для сервера может понадобиться установить диспетчер безопасности.

Форматирование текста

Пакет `java.text` позволяет форматировать, производить поиск и манипулировать текстом. В главе 33 продемонстрирован класс `NumberFormat`, который используется для форматирования числовых данных. В этом разделе рассматриваются два наиболее часто используемых класса, предназначенные для форматирования даты и времени.

Класс `DateFormat`

Класс `DateFormat` является абстрактным классом, с помощью которого можно форматировать и анализировать показания даты и времени. Метод `getDateInstance()` возвращает экземпляр класса `DateFormat`, который может форматировать информацию о дате. Можно использовать одну из следующих его форм.

```
static final DateFormat getDateInstance()
static final DateFormat getDateInstance(int стиль)
static final DateFormat getDateInstance(int стиль, Locale регион)
```

Параметр *стиль* может принимать следующие значения: DEFAULT, SHORT, MEDIUM, LONG или FULL. Они представляют собой константы типа int, определяемые в классе DateFormat. С их помощью можно представлять различные подробные сведения, связанные с датой. Параметр *регион* представляет одну из статических ссылок, определяемых классом Locale (см. главу 18). Если параметры *стиль* и/или *регион* не будут заданы, используются значения, принятые по умолчанию.

Одним из наиболее часто используемых методов в этом классе является format(). Он имеет несколько перегруженных форм, среди которых можно выделить следующую.

```
final String format(Date d)
```

Параметром является объект класса Date, который необходимо отобразить. Метод возвращает строку, содержащую отформатированную информацию.

В следующем коде показано, как производится форматирование информации о дате. Он начинается с создания объекта класса Date. Этот объект хранит информацию о текущих дате и времени. Затем он выводит информацию о дате с использованием различных стилей и с учетом местной специфики представления времени.

```
// Иллюстрация 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("Великобритания: " + df.format(date));

        df = DateFormat.getDateInstance(DateFormat.FULL, Locale.US);
        System.out.println("США: " + df.format(date));
    }
}
```

Ниже представлен пример результатов выполнения этой программы.

```
Япония: 11/01/01
Корея: 2011. 1. 1
Великобритания: 01 January 2011
США: Saturday, January 1, 2011
```

Метод getTimeInstance() возвращает экземпляр класса DateFormat, который может форматировать информацию о времени. Он имеет следующие варианты.

```
static final DateFormat getTimeInstance()
static final DateFormat getTimeInstance(int стиль)
static final DateFormat getTimeInstance(int стиль, Locale регион)
```

Параметр *стиль* принимает одно из следующих значений: DEFAULT, SHORT, MEDIUM, LONG и FULL. Они являются константами типа int, определяемыми

в классе DateFormat. С их помощью можно определить, насколько подробным будет представление времени. Параметр *регион* является одной из статических ссылок, определенных в классе Locale. Если параметры *стиль* и *регион* не будут заданы, используются значения, принятые по умолчанию.

В следующем коде показано, как производится форматирование информации о времени. Он начинается с создания объекта класса Date, который получает информацию о текущих дате и времени, а затем выводит информацию о времени, используя различные стили и местную специфику представления времени.

```
// Иллюстрация 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("Великобритания: " + df.format(date));
        df = DateFormat.getTimeInstance(DateFormat.FULL, Locale.CANADA);
        System.out.println("Канада: " + df.format(date));
    }
}
```

Ниже показан пример результатов выполнения этой программы.

```
Япония: 20:25
Великобритания: 20:25:14 CDT
Канада: 8:25:14 o'clock PM CDT
```

Класс DateFormat имеет также метод getDateTimeInstance(), который может форматировать информацию о дате и времени. Если хотите, поэкспериментируйте с ним самостоятельно.

Класс SimpledateFormat

Класс SimpleDateFormat является конкретным подклассом класса DateFormat. Он позволяет определять собственные шаблоны форматирования, которые можно будет использовать для отображения информации о дате и времени.

Ниже показан один из его конструкторов.

```
SimpleDateFormat(String строкаФормата)
```

Параметр *строкаФормата* описывает способ отображения даты и времени. Ниже показан один из примеров его применения.

```
SimpleDateFormat sdf = SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
```

Символы, используемые в строке форматирования, определяют отображаемую информацию. В табл. 28.5 приводятся эти символы и дается описание каждого из них.

Таблица 28.5. Символы форматирования строки для класса SimpleDateFormat

Символ	Описание
A	AM или PM
D	День месяца (1–31)

Окончание табл. 28.5

Символ	Описание
H	Часы в формате AM/PM (1–12)
K	Часы в формате суток (1–24)
M	Минуты (0–59)
S	Секунды (0–59)
W	Неделя в году (1–52)
Y	Год
Z	Часовой пояс
D	День в году (1–366)
E	День недели (например, Thursday)
F	День недели в месяце
G	Эпоха (т.е. AD — после Рождества Христова, BC — до Рождества Христова)
H	Часы в сутках (0–23)
K	Часы в формате AM/PM (0–11)
M	Месяц
S	Миллисекунды в секунде
W	Неделя в месяце (1–5)
Z	Часовой пояс в формате, описанном в документе RFC 822

В большинстве случаев количество повторений символа определяет способ представления даты. Текстовая информация отображается в виде аббревиатуры, если буква шаблона повторяется менее четырех раз. В противном случае используется форма без применения аббревиатуры. Например, шаблон zzzz может отобразить Pacific Daylight Time (тихоокеанское время), а шаблон zzz — PDT.

Для чисел количество повторений буквы шаблона определяет количество цифр в представлении. Например, hh:mm:ss может представить 01:51:15, в то время как h:m:s показывает то же время в виде 1:51:15.

И наконец, M или MM отвечает за отображение месяца в виде одной или двух цифр. Три или более повторений M приведут к тому, что месяц будет отображаться в виде текстовой строки.

В следующей программе демонстрируется использование этого класса.

```
// Демонстрация применения 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");
```

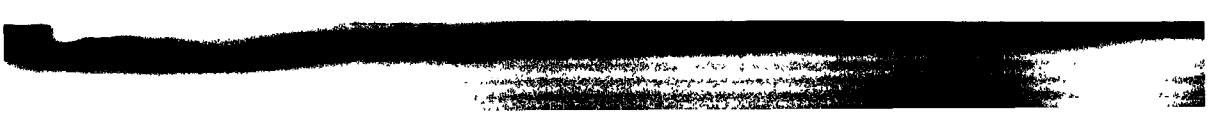
```
        System.out.println(sdf.format(date));
        sdf = new SimpleDateFormat("E MMM dd yyyy");
        System.out.println(sdf.format(date));
    }
}
```

Ниже приведен пример результатов выполнения этой программы.

```
12:46:49
01 Jan 2011 12:46:49 CST
Sat Jan 01 2011
```

ЧАСТЬ III

Разработка программного обеспечения с использованием Java



ГЛАВА 29

Компоненты Java Beans

В этой главе представлен обзор компонентов Java Beans. Важность этих компонентов бесспорна, так как они позволяют строить сложные системы на основе программных компонентов. Эти компоненты можно создавать самостоятельно, а также приобретать у сторонних разработчиков. Компоненты Java Beans определяют архитектуру, устанавливающую порядок взаимодействия строительных блоков.

Чтобы легче было понять, что представляют собой компоненты Java Beans, рассмотрим пример. У разработчиков компьютерного оборудования имеется множество компонентов, которые можно интегрировать друг с другом для построения системы. Резисторы, конденсаторы и катушки индуктивности являются примерами простых компоновочных блоков. Интегральные схемы предлагают еще больше функциональных возможностей. При этом каждая из этих частей пригодна для многократного использования. Их не нужно компоновать заново каждый раз, когда возникает необходимость в новой системе. А еще эти же элементы могут использоваться в схемах различных типов. Это возможно благодаря тому, что поведение данных компонентов понятно и хорошо документировано.

В индустрии программного обеспечения также пытались воспользоваться преимуществами многократного использования компонентов и их способностью к взаимодействию. Для этого необходимо было разработать такую компонентную архитектуру, которая позволила бы собирать программы на основе программных компоновочных блоков, пусть даже и предлагаемых сторонними разработчиками. Кроме того, проектировщик должен был иметь возможность выбрать компонент, разобраться с его функциями и внедрить в приложение. С появлением новой версии компонента необходимо, чтобы его функциональные возможности можно было без труда внедрять в существующий код. К счастью, такую архитектуру как раз и предлагают компоненты Java Beans.

Что такое Java Beans

Java Beans — это компонент программного обеспечения, предназначенный для многократного использования во множестве различных сред. Компонент Java Beans не имеет ограничений в плане функциональных возможностей. Он может выполнять простую функцию, например получать стоимость товарно-материальных запасов, а может реализовать и более сложную функцию, например предсказывать котировку акций компании. Компонент Java Beans может быть видимым для конечного пользователя. Одним из примеров такого компонента является кнопка графического интерфейса пользователя. Компонент Java Beans может быть также и невидимым для пользователя. Таким компоновочным блоком является ПО для декодирования потока мультимедийной информации в реальном времени. И наконец, компонент Java Beans может быть предназначен для работы в автономном режиме на рабочей

станции пользователя или в комплексе с другими распространяемыми компонентами. Примером компонента Java Beans, который может работать локально, является ПО для создания секторной диаграммы на основе набора точек данных. А, например, компонент Java Beans, предоставляющий информацию о ценах на фондовой или товарной бирже в реальном времени, может работать только в комплексе с другим распространяемым ПО, получая от него необходимые данные.

Преимущества компонентов Java Beans

В следующем списке перечислены некоторые преимущества, которые предлагает технология компонентов Java Beans разработчику компонентов.

- Компонент Java Beans обладает всеми преимуществами парадигмы Java, гласящей следующее: "написано однажды, работает везде".
- Можно управлять свойствами, событиями и методами компонента Java Beans, доступными другому приложению.
- Для конфигурирования компонента Java Beans можно применять вспомогательное ПО. Оно необходимо только при настройке параметров времени проектирования данного компонента. В среду выполнения его включать не нужно.
- Настройки параметров конфигурации компонента Java Beans можно хранить на постоянном носителе информации и восстанавливать по мере необходимости.
- Компонент Java Beans может регистрироваться на получение извещений о событиях от других объектов и может извещать о событиях, происходящих в других объектах.

Самодиагностика

В основе компонентов Java Beans лежит свойство *самодиагностики* (introspection). Самодиагностика — это процесс анализа компонента Java Beans, при котором определяются его характеристики. На самом деле это очень важная особенность API Java Beanss, поскольку с ее помощью другое приложение, например инструмент разработки, может получать информацию о данном компоненте. Без самодиагностики технология компонентов Java Beans работать не будет.

Существует два способа, с помощью которых разработчик компонента Java Beans может показать, какие его свойства, события и методы будут доступны. В одном случае используются простые соглашения об именовании. Они позволяют механизмам самодиагностики логически выводить информацию о компоненте Java Beans. В другом случае применяется дополнительный класс, расширяющий интерфейс `BeansInfo`, который явным образом предоставляет эту информацию. Рассмотрением этих способов мы сейчас и займемся.

Проектные шаблоны для свойств

Свойство (property) компонента Java Beans представляет собой сокращенный вариант его состояния. Значения, присваиваемые свойствам, определяют поведение и появление этого компонента. Настройка свойства осуществляется при помощи *метода записи* (setter method), а его получение — при помощи *метода чтения* (getter method). Свойства бывают двух видов: простые и индексированные.

Простые свойства

Простое свойство имеет одно значение. Его можно идентифицировать с помощью следующих проектных шаблонов, в которых *N* — это имя свойства, а *T* — его тип.

```
public T getN( );
public void setN(T арг);
```

Каждый метод имеет свойство чтения и записи для доступа к своим значениям. Свойство только для чтения имеет только метод `get`. Свойство только для записи имеет только метод `set`.

Ниже представлен пример трех простых свойств чтения/записи, а также их методов чтения и записи.

```
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;
}
```

Индексированные свойства

Индексированное свойство состоит из нескольких значений. Его можно идентифицировать с помощью следующих проектных шаблонов, в которых *N* — это имя свойства, а *T* — его тип.

```
public T getN(int индекс);
public void setN(int индекс, T значение);
public T[] getN( );
public void setN(T значения[]);
```

Ниже показан пример индексированного свойства по имени `data`, а также его методов чтения и записи.

```
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);
}
```

Проектные шаблоны для событий

Компоненты Java Beans используют модель делегирования событий, которая уже была рассмотрена в этой книге. Они способны извещать о событиях другие объекты. События могут идентифицироваться с помощью следующих проектных шаблонов, в которых *T* представляет тип события.

```
public void addTListener(TListener слушательСобытия)
public void addTListener(TListener слушательСобытия)
    throws java.util.TooManyListenersException
public void removeTListener(TListener слушательСобытия)
```

Эти методы используются для того, чтобы добавить или удалить слушателя для указанного события. Вариант метода `AddTListener()`, который не передает исключение, можно применять для *групповой передачи* события. Под групповой передачей понимается то, что на получение извещений о событии может быть зарегистрировано несколько слушателей. Вариант метода, создающего исключение `TooManyListenersException`, передает извещение о событии *индивидуально* — извещение о нем может получить только один слушатель. В любом случае метод `removeTListener()` служит для удаления слушателя. Например, если предположить, что существует тип интерфейса события, называемый `TemperatureListener`, то компонент Java Beans, который следит за температурой, может предложить следующие методы.

```
public void addTemperatureListener(TemperatureListener tl) {
    ...
}
public void removeTemperatureListener(TemperatureListener tl) {
    ...
}
```

Методы и проектные шаблоны

Проектные шаблоны не используются для именования методов, не связанных со свойствами. Механизм самодиагностики находит все открытые методы компонента Java Beans. Защищенные и закрытые методы остаются недоступными.

Использование интерфейса `BeansInfo`

Как было сказано выше, проектные шаблоны *неявно* определяют, какая информация является доступной пользователю компонента Java Beans. Интерфейс `BeansInfo` позволяет *явно* управлять доступом к информации. Интерфейс `BeansInfo` определяет несколько методов, включая перечисленные ниже.

`PropertyDescriptor[]`	`getPropertyDescriptors()`
`EventSetDescriptor[]`	`getEventSetDescriptors()`
`MethodDescriptor[]`	`getMethodDescriptors()`

Эти методы возвращают массивы объектов, содержащие информацию о свойствах, событиях и методах компонента Java Beans. Классы `PropertyDescriptor`, `EventSetDescriptor` и `MethodDescriptor` определены в пакете `java.beans` и описывают указанные элементы. Реализуя эти методы, разработчик может в точности определить, что конкретно доступно пользователю, не прибегая к самодиагностике, выполняемой на основе проектных шаблонов.

При создании класса, реализующего интерфейс `BeansInfo`, его нужно называть в формате *имя*`BeansInfo`, где *имя* — имя компонента Java Beans. Например, если компонент Java Beans называется `MyBeans`, то информационный класс должен выглядеть как `MyBeansBeansInfo`.

Чтобы упростить использование интерфейса `BeansInfo`, компоненты Java Beans предлагают класс `SimpleBeansInfo`. Он обеспечивает стандартные реализации интерфейса `BeansInfo`, включая только что представленные три метода. Можно расширить этот класс и переопределить один или более методов, чтобы явно управлять доступными аспектами компонента Java Beans. Если метод не переопределить, будет использоваться самодиагностика проектного шаблона. Например, если не переопределить метод `getPropertyDescriptors()`, то для определения свойств компонента Java Beans будут применяться проектные шаблоны. Далее в этой главе вы сможете увидеть класс `SimpleBeansInfo` в действии.

Связанные и ограниченные свойства

Компонент Java Beans, имеющий *связанное* (bound) свойство, передает извещение о событии во время изменения свойства. Происходящее событие имеет тип `PropertyChangeEvent`, и извещение о нем отправляется объектам, которые предварительно зарегистрировались на получение таких уведомлений. Класс, осуществляющий обработку этого события, должен реализовать интерфейс `PropertyChangeListener`.

Компонент Java Beans, имеющий *ограниченное* (constrained) свойство, передает извещение о событии при попытке изменения его значения. Он также передает извещение о событии, имеющее тип `PropertyChangeEvent`. Это извещение о событии посылается объектам, предварительно зарегистрировавшимся на получение таких уведомлений. Однако эти объекты могут отклонить предложенное изменение, передав исключение `PropertyVetoException`. Благодаря этой особенности компонент Java Beans может работать по-разному в зависимости от среды времени выполнения. Класс, осуществляющий обработку этого события, должен реализовать интерфейс `VetoableChangeListener`.

Постоянство

Постоянство (persistence) — это способность сохранять текущее состояние компонента Java Beans (включая значения свойств компонентов Java Beans и переменные экземпляров) на энергонезависимом запоминающем устройстве и извлекать его по мере необходимости.

Для обеспечения постоянства компонентов Java Beans используются возможности сериализации, которые предлагают библиотеки классов Java.

Самый простой способ сериализировать компонент Java Beans заключается в том, чтобы он реализовывал интерфейс `java.io.Serializable`, который является просто маркерным интерфейсом. Реализация интерфейса `java.io.Serializable` обеспечит автоматическое выполнение сериализации. Вашему компоненту Java Beans не нужно будет предпринимать никаких других действий. Автоматическую сериализацию также можно наследовать. Другими словами, если какой-то суперкласс компонента Java Beans будет реализовывать интерфейс `java.io.Serializable`, он приобретет возможность автоматической сериализации. Существует одно важное ограничение: любой класс, реализующий интерфейс `java.io.Serializable`, должен предоставлять конструктор без параметров.

При автоматической сериализации можно выборочно отключить сохранение отдельного поля с помощью ключевого слова `transient`. Таким образом, элементы данных компонента Java Beans, определенные как `transient`, сериализироваться не будут.

Если компонент Java Beans не реализует интерфейс `java.io.Serializable`, вы должны обеспечить возможность сериализации самостоятельно (например, с помощью интерфейса `java.io.Externalizable`). В противном случае контейнеры не смогут хранить конфигурацию вашего компонента.

Конфигураторы

Разработчик компонентов Java Beans может предусмотреть возможность использования *конфигуратора* (customizer), с помощью которого другой разработчик сможет конфигурировать эти компоненты Java Beans. Конфигуратор может обеспечивать поэтапное руководство всем процессом конфигурирования, выполняя указания которого можно добиться того, чтобы компонент использовался в определенном контексте. Можно также предоставить интерактивную документацию. У разработчика компонентов Java Beans будет достаточно возможностей для того, чтобы разработать такой конфигуратор, который сможет обособить его продукт на рынке.

API Java Beans

Функциональные возможности компонентов Java Beans обеспечиваются классами и интерфейсами пакета `java.beans`. В этом разделе приводится краткий обзор содержимого данного пакета. В табл. 29.1 представлен список интерфейсов пакета `java.beans` с кратким описанием их функциональных возможностей. В табл. 29.2 приведен список классов пакета `java.beans`.

Таблица 29.1. Интерфейсы пакета `java.beans`

Интерфейс	Описание
`AppletInitializer`	Методы этого интерфейса используются для инициализации компонентов Java Beans, которые также являются аплетами
`BeansInfo`	Этот интерфейс позволяет разработчику определять информацию о свойствах, событиях и методах компонента Java Beans

Окончание табл. 29.1

Интерфейс	Описание
`Customizer`	Этот интерфейс позволяет разработчику предоставить графический интерфейс пользователя, при помощи которого можно конфигурировать компонент Java Beans
`DesignMode`	Методы этого интерфейса определяют, выполняется ли компонент Java Beans в режиме проектирования
`ExceptionListener`	Метод этого интерфейса вызывается во время возникновения исключения
`PropertyChangeListener`	Метод этого интерфейса вызывается при изменении ограниченного свойства
`PropertyEditor`	Объекты, реализующие этот интерфейс, позволяют разработчикам изменять и отображать значения свойств
`VetoableChangeListener`	Метод этого интерфейса вызывается при изменении ограниченного свойства
`Visibility`	Методы этого интерфейса позволяют компоненту Java Beans работать в тех средах, в которых нет графического интерфейса пользователя

Таблица 29.2. Классы пакета `java.beans`

Класс	Описание
`BeansDescriptor`	Этот класс предлагает информацию о компоненте Java Beans. Он также позволяет связывать конфигуратор с компонентом Java Beans
`Beans`	Этот класс используется для получения информации о компоненте Java Beans
`DefaultPersistenceDelegate`	Подкласс класса `PersistenceDelegate`
`Encoder`	Зашифровывает состояние совокупности компонентов Java Beans. Может использоваться для записи этой информации в поток
`EventHandler`	Поддерживает динамическое создание слушателя событий
`EventSetDescriptor`	Экземпляры этого класса описывают событие, которое может создаваться компонентом Java Beans
`Expression`	Инкапсулирует вызов метода, возвращающего результат
`FeatureDescriptor`	Суперкласс классов `Propertydescriptor`, `EventSetDescriptor` и `MethodDescriptor`
`IndexedPropertyChangeEvent`	Подкласс класса `PropertyChangeEvent`, представляющий изменение индексированного свойства
`IndexedPropertyDescriptor`	Экземпляры этого класса описывают индексированное свойство компонента Java Beans
`IntrospectionException`	Выражение этого типа создается, если во время анализа компонента Java Beans возникает ошибка
`Introspector`	Этот класс анализирует компонент Java Beans и создает объект `BeansInfo`, описывающий компонент
`MethodDescriptor`	Экземпляры этого класса описывают метод компонента Java Beans

Окончание табл. 29.2

Класс	Описание
`ParameterDescriptor`	Экземпляры этого класса описывают параметр метода
`PersistenceDelegate`	Обрабатывает информацию о состоянии объекта
`PropertyChangeEvent`	Это событие происходит при изменении связанных или ограниченных свойств. Извещение о нем посылается объектам, которые зарегистрировались на получение извещений об этих событиях и реализуют один из интерфейсов `PopertyChangeListener` и `VetoableChangeListener`
`PropertyChangeListenerProxy`	Расширяет класс `EventListenerProxy` и реализует интерфейс `PropertyChangeListener`
`PropertyChangeSupport`	Компоненты Java Beans, поддерживающие ограниченные свойства, могут использовать этот класс для уведомления объектов интерфейса `PropertyChangeListener`
`PropertyDescriptor`	Экземпляры этого класса описывают свойство компонента Java Beans
`PropertyEditorManager`	Этот класс обнаруживает объект `PropertyEditor` для данного типа
`PropertyEditorSupport`	Этот класс предлагает функциональные возможности, которые могут использоваться при написании редакторов свойств
`PropertyVetoException`	Исключение данного типа создается при отклонении изменения ограниченного свойства
`SimpleBeansInfo`	Этот класс предлагает функциональные возможности, которые могут использоваться при написании классов `BeansInfo`
`Statement`	Инкапсулирует вызов метода
`VetoableChangeListenerProxy`	Расширяет класс `EventListenerproxy` и реализует интерфейс `VetoableChangeListener`
`VetoableChangeSupport`	Компоненты Java Beans, которые поддерживают ограниченные свойства, могут использовать этот класс для уведомления объектов интерфейса `VetoableChangeListener`
`XMLDecoder`	Используется для чтения компонента Java Beans из документа XML
`XMLEncoder`	Используется для записи компонента Java Beans в документ XML

Хотя эта глава не позволяет рассказать обо всех классах, мы поговорим о четырех из них, представляющих определенный интерес, — `Introspector`, `PropertyDescriptor`, `EventSetDescriptor` и `MethodDescriptor`.

Класс `Introspector`

Этот класс предлагает несколько статических методов, поддерживающих самодиагностику. Наиболее интересным из них является метод `getBeansinfo()`. Он возвращает объект `BeansInfo`, который может использоваться для получения информации о компоненте Java Beans.

Метод `getBeansInfo()` имеет несколько форм, включая следующую.

```
static BeansInfo getBeansInfo(Class<?> Beans) throws
IntrospectionException
```

Возвращаемый объект содержит информацию о заданном компоненте Java Beans.

Класс `PropertyDescriptor`

Класс `PropertyDescriptor` описывает свойство компонента Java Beans. Он поддерживает несколько методов, которые управляют свойствами и описывают их. Например, вы можете определить, ограничено ли свойство, вызовом функции `isBound()`. Чтобы определить, является ли свойство ограниченным, нужно вызвать функцию `isConstrained()`. Имя свойства можно получить при помощи вызова функции `getName()`.

Класс `EventSetDescriptor`

Класс `EventSetDescriptor` представляет событие компонента Java Beans. Он поддерживает несколько методов, которые получают методы, используемые компонентом Java Beans для добавления/удаления слушателей событий или для управления событиями. Например, чтобы получить метод, служащий для добавления слушателей, нужно вызвать метод `getAddListenerMethod()`. Чтобы получить метод, используемый для удаления слушателей, необходимо вызвать метод `getRemoveListenerMethod()`. Чтобы получить тип слушателя, следует вызвать метод `getListenerType()`. Имя события можно получить, если вызвать метод `getName()`.

Класс `MethodDescriptor`

Класс `MethodDescriptor` представляет метод компонента Java Beans. Чтобы получить имя метода, нужно вызвать метод `getName()`. Информацию о методе можно получить с помощью метода `getMethod()`, который показан ниже.

```
Method getMethod()
```

При этом возвращается объект класса `Method`, описывающий данный метод.

Пример компонента Java Beans

В завершение главы предлагается пример, иллюстрирующий различные аспекты программирования компонентов Java, включая самодиагностику и использование класса `BeansInfo`. В нем также участвуют классы `Introspector`, `PropertyDesciptor` и `EventSetDescriptor`. В примере используется три класса. Первый из них, являющийся компонентом Beans `Colors`, показан ниже.

```
/// Простой компонент Java Beans.
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

public class Colors extends Canvas implements Serializable {
    transient private Color color; // непостоянный
    private boolean rectangular;   // постоянный

    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);
        }
    }
}
```

Компонент Beans `Colors` отображает цветной объект в рамке. Цвет компонента определяется закрытой переменной `color` типа `Color`, а его форма — закрытой переменной `rectangular` типа `boolean`. Конструктор определяет анонимный внутренний класс, расширяющий класс `MouseAdapter`, и переопределяет метод `mousePressed()`. Метод `change()` вызывается в ответ на щелчок кнопкой мыши. Он выбирает случайный цвет и перекрашивает компонент. Методы `getRectangular()` и `setRectangular()` обеспечивают доступ к одному свойству данного компонента Java Beans. Метод `change()` вызывает метод `randomColor()` для выбора цвета, а затем вызывает метод `repaint()`, чтобы сделать изменение видимым. Обратите внимание на то, что метод `paint()` использует переменные `rectangular` и `color` для определения представления компонента Java Beans.

Следующим классом является `ColorsBeansInfo`. Это подкласс класса `SimpleBeansInfo`, который предлагает явную информацию о цвете. В нем переопределяется метод `getPropertyDescriptors()` для указания того, какие свойства будут доступны пользователю компонента Java Beans. В данном случае поль-

зователю доступно только свойство rectangular. Метод создает и возвращает объект класса PropertyDescriptor для свойства rectangular. Ниже показан используемый конструктор класса PropertyDescriptor.

```
PropertyDescriptor(String свойство, Class<?> BeansCls)
    throws IntrospectionException
```

Два параметра представляют, соответственно, имя свойства и класс компонента Java Beans.

```
// Информационный класс компонента Java Beans.
import java.beans.*;
public class ColorsBeansInfo extends SimpleBeansInfo {
    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            PropertyDescriptor rectangular = new
            PropertyDescriptor("rectangular", Colors.class);
            PropertyDescriptor pd[] = {rectangular};
            return pd;
        } catch(Exception e) {
            System.out.println("Передано исключение. " + e);
        }
        return null;
    }
}
```

Последним классом является IntrospectorDemo. Он использует самодиагностику для отображения свойств и событий, доступных в компоненте Java Beans Colors.

```
// Демонстрация свойств и событий.
import java.awt.*;
import java.beans.*;

public class IntrospectorDemo {
    public static void main(String args[]) {
        try {
            Class<?> c = Class.forName("Colors");
            BeansInfo BeansInfo = Introspector.getBeansInfo(c);

            System.out.println("Свойства:");
            PropertyDescriptor propertyDescriptor[] =
                BeansInfo.getPropertyDescriptors();
            for(int i = 0; i < propertyDescriptor.length; i++) {
                System.out.println("\t" +
                                   propertyDescriptor[i].getName());
            }

            System.out.println("События:");
            EventSetDescriptor eventSetDescriptor[] =
                BeansInfo.getEventSetDescriptors();
            for(int i = 0; i < eventSetDescriptor.length; i++) {
                System.out.println("\t" +
                                   eventSetDescriptor[i].getName());
            }
        } catch(Exception e) {
            System.out.println("Передано исключение. " + e);
        }
    }
}
```

Ниже показан вывод, полученный в результате выполнения этой программы.

```
События:
    mouseWheel
    mouse
    mouseMotion
    component
    hierarchyBounds
    focus
    hierarchy
    propertyChange
    inputMethod
    key
```

В этом выводе следует обратить внимание на следующее. Поскольку класс `ColorsBeansInfo` заменяет метод `getPropertyDescriptors()` так, что единственным возвращаемым свойством является `rectangular`, отображается только свойство `rectangular`. Однако поскольку метод `getEventSetDescriptors()` не переопределяется в классе `ColorsBeansInfo`, используется самодиагностика проектного шаблона и обнаруживаются все события, включая те, которые принадлежат классу `Colors` суперкласса `Canvas`. Помните, что если не переопределить один из методов "get", определенных в классе `SimpleBeansInfo`, то по умолчанию будет использована самодиагностика проектного шаблона. Чтобы увидеть, какие изменения были произведены классом `ColorsBeansInfo`, нужно удалить его файл класса и запустить еще раз `IntrospectorDemo`. На этот раз он предоставит отчет по большему количеству свойств.

ГЛАВА 30

Введение в библиотеку Swing

В части II настоящей книги вы могли видеть, как с помощью классов библиотеки AWT можно создавать пользовательские интерфейсы. Несмотря на то что библиотека AWT по-прежнему является важной частью Java, ее набор компонентов применяется для создания графических пользовательских интерфейсов уже не так широко. Сегодня многие программисты пользуются для этой цели классами библиотеки Swing. Библиотека Swing — это набор классов, которые предлагают более мощные и гибкие компоненты GUI, чем библиотека AWT. Вообще говоря, современный графический пользовательский интерфейс Java построен на основе библиотеки Swing.

Рассмотрению библиотеки Swing посвящено две главы. Здесь вы ознакомитесь с основами библиотеки Swing. Начнем с описания ее основных концепций. Затем будет представлена общая форма программы на базе библиотеки Swing, включая приложения и аплеты. В конце главы объясним, как с помощью библиотеки Swing можно делать рисунки. В следующей главе будет рассмотрено несколько наиболее часто используемых компонентов библиотеки Swing. Важно понять, что количество классов и интерфейсов в пакетах Swing довольно большое и что каждый из них просто невозможно рассмотреть в этой книге. (Вообще говоря, чтобы полностью рассказать о библиотеке Swing, потребуется отдельная книга.) И все же в этих двух главах вы сможете узнать о библиотеке Swing самое главное.

На заметку! Более содержательное описание библиотеки Swing можно найти в моей книге *Swing: руководство для начинающих*, перевод который на русский язык вышел в И. Д. "Вильямс" в2007 году.

Истоки библиотеки Swing

В начале существования языка Java классов библиотеки Swing не было вообще. Причиной разработки классов библиотеки Swing было слабое место в исходной подсистеме GUI Java — библиотеке AWT. Библиотека AWT определяет базовый набор элементов управления, окон и диалоговых окон, которые поддерживают пригодный к использованию, но ограниченный в возможностях графический интерфейс. Одной из причин ограниченности библиотеки AWT является то, что она преобразует свои визуальные компоненты в независящие от платформы соответствующие им эквиваленты, которые называются *равноправными компонентами*. Это означает, что внешний вид компонента определяется платформой, а не закладывается в Java. Поскольку компоненты библиотеки AWT используют "машинно-зависимые" ресурсы кода, они называются *тяжеловесными*.

Использование "машинно-зависимых" равноправных компонентов порождает некоторые проблемы. Во-первых, в связи с различиями, существующими между опера-

ционными системами, компонент может выглядеть или даже вести себя по-разному на различных платформах. Такая изменчивость шла вразрез с философией Java: "написано однажды, работает везде". Во-вторых, внешний вид каждого компонента был фиксированным (так как все зависело от платформы), и это нельзя было изменить (по крайней мере, делалось это с трудом). В-третьих, применение тяжеловесных компонентов влекло за собой появление новых ограничений. К примеру, тяжеловесный компонент всегда имеет прямоугольные очертания и является непрозрачным.

Вскоре после появления первоначальной версии Java стало очевидным, что ограничения, присущие библиотеке AWT, были настолько серьезными, что нужно было найти лучший подход. В итоге появились классы Swing как часть набора библиотек классов Java Foundation Classes (JFC). В 1997 году они были включены в Java 1.1 в виде отдельной библиотеки. А начиная с версии Java 1.2 классы Swing (а также все остальное, что входило в состав набора JFC) стали полностью интегрированными в Java.

Классы библиотеки Swing построены на основе библиотеки AWT

Прежде чем продолжить, необходимо сделать одно важное замечание: хотя библиотека Swing снимает некоторые ограничения, присущие библиотеке AWT, она *не заменяет* библиотеку AWT. Наоборот, классы библиотеки Swing построены на основе библиотеки AWT. Вот почему библиотека AWT до сих пор является важной частью языка Java. Кроме того, библиотека Swing использует тот же механизм обработки событий, что и библиотека AWT. Таким образом, перед использованием библиотеки Swing нужно разобраться с тем, как работает библиотека AWT. (О библиотеке AWT речь велась в главах 24 и 25. Обработке событий была посвящена глава 23.)

Две ключевые особенности библиотеки Swing

Как только что было сказано, классы библиотеки Swing были созданы для того, чтобы устранить ограничения, присущие библиотеке AWT. Этого удалось достичь благодаря двум ключевым особенностям: облегченным компонентам и подключаемому внешнему виду. Вместе они предлагают элегантное и простое в использовании решение проблем библиотеки AWT. Именно эти две особенности и определяют суть библиотеки Swing. О каждой из них поговорим прямо сейчас.

Компоненты библиотеки Swing являются облегченными

За некоторыми исключениями, компоненты библиотеки Swing являются *облегченными*. Это означает, что они написаны исключительно на языке Java и не преобразуются в равноправные компоненты, специфические для данной платформы. Следовательно, облегченные компоненты являются более эффективными и гибкими. Более того, поскольку облегченные компоненты не преобразуются в равноправные "машинно-зависимые" компоненты, внешний вид каждого компонента определяется классами библиотеки Swing, а не базовой операционной системой. Это означает, что каждый компонент будет работать одинаково на всех платформах.

Библиотека Swing поддерживает подключаемый внешний вид

Библиотека Swing поддерживает принцип *подключаемого внешнего вида* (Pluggable Look And Feel – PLAF). Поскольку каждый компонент библиотеки Swing визуализируется кодом Java, а не "машинно-зависимыми" равноправными компонентами, внешний вид компонента находится под контролем библиотеки Swing. Это говорит о том, что внешний вид можно отделить от логики компонента, что и делается в библиотеке Swing. В этом есть и свое преимущество: так можно изменить способ визуализации компонента, не затрагивая какой-либо из его остальных аспектов. Другими словами, можно "подключить" новый внешний вид любого заданного компонента, не создавая при этом каких-либо побочных эффектов в коде, использующем данный компонент. Более того, можно определить наборы внешних видов, которые будут представлять разные стили пользовательского графического интерфейса. Чтобы использовать определенный стиль, нужно просто "подключить" его внешний вид. Как только это будет сделано, все компоненты автоматически будут визуализироваться с помощью этого стиля.

Возможность подключаемого внешнего вида имеет несколько важных преимуществ. Например, вы можете задать параметры, которые будут одинаковыми для всех платформ. И наоборот, можно задать внешний вид, специфичный для определенной платформы. Например, если вы знаете, что приложение будет работать только в среде Windows, можно определить внешний вид для Windows. Можно также разработать и специальный внешний вид. И наконец, внешний вид можно динамически изменять во время работы приложения.

Язык Java 7 предлагает такие внешние виды, как metal, Motif и Nimbus, которые доступны всем пользователям библиотеки Swing. Внешний вид metal (металлический) также называется *внешним видом Java*. Он не зависит от платформы и доступен во всех средах, в которых работает Java. Кроме того, он еще и выбирается по умолчанию. В среде Windows можно использовать внешние виды Windows и Windows Classic. В этой книге используется внешний вид metal, поскольку он не зависит от платформы.

Архитектура MVC

В общем случае визуальный компонент определяется тремя отдельными аспектами:

- как компонент выглядит во время его визуализации на экране;
- как компонент взаимодействует с пользователем;
- информацией о состоянии компонента.

Независимо от того, какая архитектура используется для реализации компонента, она должна неявно включать эти три аспекта. Вот уже несколько лет свою исключительную эффективность доказала архитектура *"модель-представление-контроллер"* (Model-View-Controller – MVC).

Успех архитектуры MVC объясняется тем, что каждый элемент дизайна соответствует некоторому аспекту компонента. Исходя из терминологии MVC, *модель* соответствует информации о состоянии, связанной с компонентом. Например, в случае флажка модель содержит поле, которое показывает, установлен ли флажок. *Представление* определяет, как компонент отображается на экране, включая любые аспекты представления, зависящие от текущего состояния модели. *Контроллер* определяет, как компонент будет реагировать на действия пользовате-

ля. Например, когда пользователь щелкает на флажке, контроллер реагирует изменением модели, чтобы отразить выбор пользователя (отметка флажка или снятие отметки). Это впоследствии приводит к обновлению представления. Разделяя компонент на модель, представление и контроллер, можно изменять конкретную реализацию любого из этих аспектов, не затрагивая остальные. Например, различные реализации представлений могут визуализировать один и тот же компонент разными способами, однако это не будет влиять на модель или контроллер.

Хотя архитектура MVC и принципы, заложенные в ее основе, выглядят концептуально, высокий уровень разделения между представлением и контроллером не дает никаких преимуществ компонентам библиотеки Swing. Наоборот, библиотека Swing использует модифицированную версию MVC, которая объединяет представление и контроллер в один логический объект, называемый *делегатом пользовательского интерфейса* (UI delegate). В связи с этим подход, применяемый в библиотеке Swing, называется или архитектурой *"модель-делегат"* (Model-Delegate), или архитектурой *"разделяемая модель"* (Separable Model). Поэтому, несмотря на то что архитектура компонентов библиотеки Swing основана на MVC, она не использует ее классическую реализацию.

Подключаемый внешний вид в библиотеке Swing возможен благодаря архитектуре "модель-делегат". Поскольку представление и контроллер отделены от модели, внешний вид можно изменять, не влияя на способ использования компонента внутри программы. И наоборот, можно настроить модель, не влияя на способ отображения компонента на экране или реакцию на действия пользователя.

Для поддержания архитектуры "модель-делегат" большинство компонентов библиотеки Swing содержит два объекта. Один из них представляет модель, а другой — делегата пользовательского интерфейса. Модели определяются интерфейсами. Например, модель кнопки определяется интерфейсом `ButtonModel`. Делегаты пользовательского интерфейса являются классами, унаследованными от `ComponentUI`. Например, делегатом пользовательского интерфейса кнопки является `ButtonUI`. Как правило, ваши программы не будут взаимодействовать напрямую с делегатами пользовательского интерфейса.

Компоненты и контейнеры

Графический пользовательский интерфейс библиотеки Swing состоит из двух ключевых элементов: *компонентов* и *контейнеров*. Однако это в основном концептуальное разделение, так как все контейнеры тоже являются компонентами. Различие между этими двумя элементами кроется в их назначении: компонент представляет собой независимый визуальный элемент управления вроде кнопки или ползунка, а контейнер вмещает группу компонентов. Таким образом, контейнер является особым типом компонента и предназначен для содержания других компонентов. Более того, чтобы можно было отобразить компонент, он должен находиться внутри контейнера. Так, во всех пользовательских графических интерфейсах библиотеки Swing имеется как минимум один контейнер. Поскольку контейнеры являются компонентами, контейнер может содержать другие контейнеры. Благодаря этому принципу библиотека Swing может определить *иерархию вместимости*, на вершине которой должен находиться *контейнер верхнего уровня*.

А теперь давайте поближе познакомимся с компонентами и контейнерами.

Компоненты

В общем случае компоненты библиотеки Swing происходят от класса `JComponent`. (Исключением является четыре контейнера верхнего уровня, о которых речь пойдет в следующем разделе.) Класс `JComponent` предлагает функциональные возможности, общие для всех компонентов. Например, класс `JComponent` поддерживает подключаемый внешний вид. Класс `JComponent` наследует классы библиотеки AWT `Container` и `Component`. Следовательно, компонент библиотеки Swing основан на компоненте библиотеки AWT и совместим с ним.

Все компоненты библиотеки Swing представлены классами, определенными в пакете `javax.swing`. Ниже перечислены имена классов компонентов библиотеки Swing (включая компоненты, используемые в качестве контейнеров).

`JApplet`	`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 на то, что все классы компонентов начинаются с буквы `J`. Например, классом для метки является `JLabel`, классом для кнопки — `JButton`, классом для ползунка — `JScrollBar`.

Контейнеры

В библиотеке Swing определены два типа контейнеров. Первый тип — это контейнеры верхнего уровня. К ним относятся контейнеры классов `JFrame`, `JApplet`, `JWindow` и `JDialog`. Классы этих контейнеров не являются наследниками класса `JComponent`. Однако они наследуют классы библиотеки AWT `Component` и `Container`. В отличие от остальных компонентов библиотеки Swing, которые являются облегченными, компоненты верхнего уровня являются тяжеловесными. Поэтому в библиотеке компонентов библиотеки Swing они представляют собой частный случай.

Судя по названию, контейнер верхнего уровня должен находиться на вершине иерархии контейнеров. Контейнер верхнего уровня не содержится ни в каком другом контейнере. Более того, каждая иерархия вместимости должна начинаться с контейнера верхнего уровня. Таким контейнером в приложениях чаще всего является класс `JFrame`. В аплетах чаще всего используется класс `JApplet`.

Вторым типом контейнеров, которые поддерживает библиотеку Swing, являются облегченные контейнеры. Они наследуются от класса `JComponent`. Примером облегченного контейнера является класс `JPanel`, который является контейнером общего назначения. Облегченные контейнеры часто используются для организации и управления группами связанных компонентов, поскольку облегченный контейнер может находиться внутри другого контейнера. Следовательно, вы можете применять облег-

ченные контейнеры наподобие класса `JPanel` для создания подгрупп связанных элементов управления, содержащихся внутри внешнего контейнера.

Панели контейнеров верхнего уровня

Каждый контейнер верхнего уровня определяет набор *панелей* (pane). Вверху иерархии находится экземпляр класса `JRootPane`. Класс `JRootPane` – это облегченный контейнер, предназначенный для управления остальными панелями. Он также помогает управлять необязательной полосой меню. Панели, содержащие корневую панель, называются *"стеклянной" панелью* (glass pane), *панелью содержимого* (content pane) и *многослойной панелью* (layered pane).

"Стеклянная" панель является панелью верхнего уровня. Она находится над всеми панелями и покрывает каждую из них. По умолчанию эта панель является прозрачным экземпляром класса `JPanel`. "Стеклянная" панель позволяет управлять событиями мыши, влияющими на весь контейнер (а не на отдельный элемент управления), или, к примеру, рисовать поверх любого другого компонента. В большинстве случаев вам не нужно будет использовать "стеклянную" панель напрямую, однако если она вам понадобится, то искать ее нужно именно здесь.

Многослойная панель является экземпляром класса `JLayeredPane`. Она позволяет задать определенную глубину размещения компонентов. Значение, соответствующее этой глубине, определяет, какой компонент перекрывает собой другой компонент. (В связи с этим многослойные панели позволяют задавать упорядоченность компонентов по координате Z, хотя это не всегда бывает необходимо.) Многослойная панель содержит панель содержимого и (необязательно) полосу меню.

Несмотря на то что "стеклянная" и многослойная панели являются неотъемлемыми частями контейнера верхнего уровня и служат для разных целей, большая часть того, что они предлагают, скрыта от пользователей. Ваше приложение будет работать чаще всего с панелью содержимого, поскольку именно на нее вы будете добавлять визуальные компоненты. Другими словами, добавляя компонент (например, кнопку) в контейнер верхнего уровня, вы добавляете его в панель содержимого. По умолчанию панель содержимого является непрозрачным экземпляром класса `JPanel`.

Пакеты библиотеки Swing

Библиотека Swing – это очень большая подсистема, в которой задействовано большое количество пакетов. Вот пакеты, определенные библиотекой Swing на момент написания этой книги.

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

Самым главным пакетом является `javax.swing`. Его нужно импортировать в любую программу, использующую библиотеку Swing. В этом пакете содержатся

классы, реализующие базовые компоненты библиотеки Swing, такие как кнопки, метки и флажки.

Простое приложение Swing

Программы Swing отличаются и от консольных программ, и от программ AWT, показанных в этой книге. Например, в отличие от библиотеки AWT, они используют другие наборы компонентов и другие иерархии контейнеров. Программы Swing имеют также особые требования, которые относятся к потоковой обработке. Самый лучший способ понять структуру программы Swing — испытать в деле рабочий пример. Существует два типа программ Java, в которых обычно используется библиотека Swing. Первый тип — это настольные приложения, а второй тип — аплеты. В этом разделе будет рассмотрен пример создания приложения Swing. Созданием аплета Swing мы тоже будем заниматься в этой главе, но чуть позже.

Несмотря на то что следующая программа довольно небольшая, она демонстрирует один из способов написания приложения Swing. Кроме того, она иллюстрирует несколько ключевых возможностей библиотеки Swing. В ней используется два компонента библиотеки Swing: `JFrame` и `JLabel`. Класс `JFrame` — контейнер верхнего уровня, который обычно используется в приложениях Swing. Класс `JLabel` — компонент библиотеки Swing, создающий метку, которая является компонентом, отображающим информацию. Метка — это самый простой компонент библиотеки Swing, поскольку она является пассивным компонентом. То есть метка не реагирует на действия пользователя. Она служит для отображения выходных данных. Программа использует контейнер класса `JFrame` для хранения экземпляра класса `JLabel`. Метка отображает короткое текстовое сообщение.

```
// Простое приложение Swing.

import javax.swing.*;

class SwingDemo {

    SwingDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("A Simple Swing Application");
     // JFrame jfrm = new JFrame("Простое приложение Swing");

        // Задаем фрейму исходный размер.
        jfrm.setSize(275, 100);

        // Прекращаем работу программы, если пользователь
        // закрывает приложение.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создаем метку с текстом.
        JLabel jlab = new JLabel("Swing means powerful GUIs.");
     // JLabel jlab = new JLabel("Swing означает мощный GUI.");

        // Добавляем метку на панель содержимого.
        jfrm.add(jlab);

        // Отображаем фрейм.
        jfrm.setVisible(true);
        }

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

```
            // Создаем фрейм в потоке диспетчеризации событий.
            SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new SwingDemo();
            }
        });
    }
}
```

Программы Swing компилируются и выполняются точно таким же способом, как и остальные приложения Java. Поэтому чтобы скомпилировать эту программу, можно воспользоваться следующей командной строкой.

```
javac SwingDemo.java
```

Чтобы запустить программу, нужно выполнить следующую команду.

```
java SwingDemo
```

Когда программа начнет работу, она создаст окно, показанное на рис. 30.1.

Поскольку программа `SwingDemo` иллюстрирует несколько основополагающих концепций библиотеки Swing, рассмотрим ее тщательно, строка за строкой. Программа начинается с импорта пакета `javax.swing`. Как уже упоминалось, этот пакет содержит компоненты, определяемые библиотекой Swing. Например, пакет `javax.swing` определяет классы, реализующие метки, кнопки, текстовые элементы управления и меню. Он будет включен во все программы, использующие библиотеку Swing.

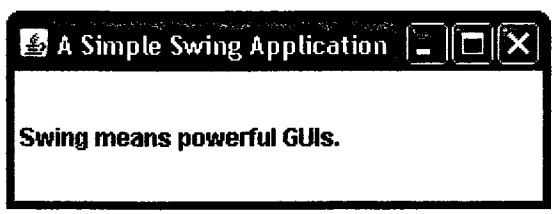


Рис. 30.1. Окно, созданное программой `SwingDemo`

Затем объявляется класс `SwingDemo` и конструктор для этого класса. Конструктор — это метод, в котором выполняется большинство действий программы. Он начинается с создания экземпляра класса `JFrame` с помощью следующей строки кода.

```
JFrame jfrm = new JFrame("A Simple Swing Application");
```

В результате будет создан контейнер `jfrm`, определяющий прямоугольное окно со строкой заголовка, кнопками закрытия, свертывания, развертывания и восстановления, а также с системным меню. Таким образом, программа создает стандартное окно верхнего уровня. Конструктору передается заголовок окна.

Затем задаются размеры окна с помощью следующего оператора.

```
jfrm.setSize(275, 100);
```

Метод `setSize()` (он наследуется классом `JFrame` от класса AWT `Component`) задает размеры окна, определяемые в пикселях. Он имеет такую форму.

`void setSize(int` *ширина*`, int` *высота*`)`

В этом примере ширина окна (*ширина*) равняется 275 пикселям, а высота (*высота*) — 100.

По умолчанию, когда закрывается окно верхнего уровня (например, когда пользователь щелкает на кнопке закрытия), окно удаляется с экрана, но работа приложения не прекращается. Несмотря на то что это поведение в некоторых си-

туациях является полезным, для большинства приложений оно не подходит. Чаще всего при закрытии окна верхнего уровня нужно будет просто прекращать работу всего приложения. Это можно сделать двумя способами. Самый простой из них — вызов метода `setDefaultCloseOperation()`, что и делается в программе.

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

Теперь работа всего приложения будет прекращаться при закрытии окна. Общая форма метода `setDefaultCloseOperation()` выглядит так.

```
void setDefaultCloseOperation(int что)
```

Значение параметра *что* определяет, что происходит при закрытии окна. Помимо значения `JFrame.EXIT_ON_CLOSE`, доступно еще несколько значений.

```
DISPOSE_ON_CLOSE
HIDE_ON_CLOSE
DO_NOTHING_ON_CLOSE
```

В их именах отражены выполняемые действия. Эти константы объявлены в классе `WindowConstants`, который является интерфейсом, объявленным в пакете `javax.swing`, реализуемым контейнером класса `JFrame`.

```
JLabel jlab = new JLabel("Swing means powerful GUIs.");
```

Класс `JLabel` — самый простой и легкий в использовании компонент, так как он не принимает от пользователя входных данных. Он просто отображает информацию, которая может представлять текст, значок или их комбинацию. Метка, созданная программой, содержит только текст, который передается ее конструктору.

Следующая строка кода добавляет метку в панель содержимого фрейма.

```
jfrm.add(jlab);
```

Как было сказано ранее, все контейнеры верхнего уровня имеют панель содержимого, в которой размещаются компоненты. Таким образом, чтобы добавить компонент во фрейм, нужно добавить его в панель содержимого фрейма. Это можно сделать, вызвав метод `add()` для ссылки на экземпляр класса `JFrame` (в данном случае контейнер `jfrm`). Общая форма метода `add()` показана ниже.

```
Component add(Component компаратор)
```

Метод `add()` наследуется классом `JFrame` от класса `Container` библиотеки AWT.

По умолчанию панель содержимого, связанная с компонентом класса `JFrame`, использует граничную компоновку. Только что показанный вариант метода `add()` добавляет метку и помещает ее в центре. Другие варианты метода `add()` позволяют задать одну из граничных областей. Когда компонент добавляется в центр, его размер подгоняется автоматически таким образом, чтобы компонент смог уместиться в центре.

Прежде чем продолжить, нужно сделать одно важное замечание. До выхода комплекта JDK 5 при добавлении компонента на панель содержимого нельзя было вызывать метод `add()` непосредственно для экземпляра класса `JFrame`. Вместо этого нужно было вызывать метод `add()` для панели содержимого объекта класса `JFrame`. Панель содержимого можно было получить в результате вызова метода `getContentPane()` для экземпляра класса `JFrame`. Метод `getContentPane()` показан ниже.

```
Container getContentPane()
```

Класс `Container` получает ссылку на окно содержимого. После этого осуществляется вызов метода `add()` по этой ссылке для добавления компонента на панель содержимого. Таким образом, чтобы добавить метку `jlab` в контейнер `jfrm`, раньше нужно было использовать следующий оператор:

```
jfrm.getContentPane().add(jlab); // старый стиль
```

Здесь метод getContentPane() сначала получает ссылку на панель содержимого, после чего метод add() добавляет компонент в контейнер, присоединенный к этому окну. Эту же процедуру нужно было выполнять для вызова метода remove(), когда требовалось удалить компонент, и метода setLayout(), чтобы задать диспетчер компоновки для окна содержимого. В коде, написанном на языке Java до версии 5.0, нередко встречаются вызовы метода getContentPane(). Однако теперь использовать этот метод больше не нужно. Можно просто вызывать методы add(), remove() и setLayout() непосредственно для объекта класса JFrame, так как они были изменены специально для того, чтобы автоматически работать с окном содержимого.

Последний оператор в конструкторе класса SwingDemo нужен для того, чтобы сделать окно видимым.

```
jfrm.setVisible(true);
```

Метод setVisible() наследуется от класса библиотеки AWT Component. Если его аргумент будет равен true, то окно будет отображаться. В противном случае оно будет скрыто. По умолчанию контейнер класса JFrame является невидимым, поэтому, чтобы показать его, нужно вызвать метод setVisible(true).

Внутри метода main() создается объект класса SwingDemo, который отображает окно и метку. Обратите внимание на то, что конструктор класса SwingDemo вызывается с помощью следующих трех строк кода.

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new SwingDemo();
    }
});
```

Выполнение этой последовательности кода приводит к созданию объекта класса SwingDemo в *потоке диспетчеризации событий*, а не в главном потоке приложения. И вот почему. В общем случае программы Swing управляются событиями. Например, когда пользователь взаимодействует с компонентом, происходит событие. Извещение о событии передается в приложение вызовом обработчика событий, определенного в приложении. Однако обработчик выполняется в потоке диспетчеризации событий, поддерживаемом библиотекой Swing, а не в главном потоке приложения. Таким образом, хотя обработчики событий и определены в программе, они вызываются в потоке, который не был создан вашей программой.

Чтобы избежать этой проблемы (включая вероятность возникновения взаимной блокировки), все компоненты GUI библиотеки Swing нужно создавать и обновлять из потока диспетчеризации событий, а не из главного потока приложения. А метод add() выполняется в главном потоке. Таким образом, метод main() не может напрямую наследовать объект класса SwingDemo. Наоборот, он должен создать объект класса, реализующего интерфейс Runnable, который выполняется в потоке диспетчеризации событий, и заставить этот объект создать GUI.

Чтобы код интерфейса GUI можно было создать в потоке диспетчеризации событий, необходимо использовать один из двух методов, определенных в классе SwingUtilities. Это методы invokeLater() и invokeAndWait().

```
static void invokeLater(Runnable объект)
static void invokeAndWait(Runnable объект)
throws InterruptedException, InvocationTargetException
```

Здесь *объект* — это объект класса, реализующего интерфейс Runnable, метод run() которого будет вызываться потоком диспетчеризации событий. Единственное различие между этими двумя методами заключается в том, что метод invokeLater() возвращает результат немедленно, а метод invokeAndWait() ожидает возврата результата метода obj.run(). Вы можете использовать эти методы для вызова ме-

тода, создающего GUI для вашего приложения Swing, или использовать их каждый раз, когда вам нужно будет изменить состояние GUI из кода, не выполняющегося в потоке диспетчеризации событий. Как правило, вам нужно будет использовать метод `invokeLater()`, как это было в предыдущей программе. А при создании исходного GUI для аплета вам понадобится метод `invokeAndWait()`.

Обработка событий

В предыдущем примере была показана базовая форма программы Swing, однако в ней не хватает одной важной части — обработки событий. Поскольку метка `JLabel` не принимает входных данных от пользователя, она не создает извещений о событиях, поэтому и обработка событий не была нужна. Однако остальные компоненты библиотеки Swing *реагируют* на вводимые пользователем данные, вследствие чего возникает необходимость в обработке событий, которые возникают в результате таких взаимодействий. Например, событие происходит тогда, когда таймер завершает отсчет. В любом случае обработка событий является большой частью любого приложения, построенного на основе библиотеки Swing.

Механизм обработки событий, используемый в библиотеке Swing, ничем не отличается от механизма, применяемого в библиотеке AWT. Этот подход называется *моделью* делегирования *событий*, и он рассматривался в главе 23. Во многих случаях библиотека Swing использует те же события, что и библиотека AWT, и эти события определены в пакете `java.awt.event`. События, являющиеся специфическими для библиотеки Swing, определены в пакете `javax.swing.event`.

Несмотря на то что события обрабатываются в библиотеке Swing точно так же, как и в библиотеке AWT, будет полезно рассмотреть небольшой и простой пример. В следующей программе обрабатывается событие, созданное кнопкой из класса библиотеки Swing. Результат показан на рис. 30.2.

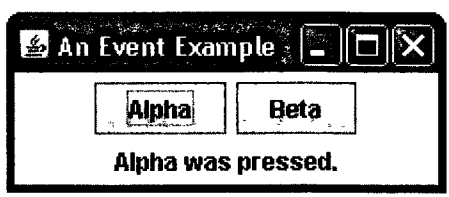


Рис. 30.2. Результат выполнения программы `EventDemo`

```
// Обработка события в программе Swing.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class EventDemo {

    JLabel jlab;

        EventDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("An Event Example");
     // JFrame jfrm = new JFrame("Пример обработки событий");
```

```
        // Определение класса FlowLayout для диспетчера компоновки.
        jfrm.setLayout(new FlowLayout());

        // Установка исходных размеров фрейма.
        jfrm.setSize(220, 90);

        // Прекращение работы программы, если пользователь
        // закрывает приложение.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создаем две кнопки.
        JButton jbtnAlpha = new JButton("Alpha");
        JButton jbtnBeta = new JButton("Beta");

        // Добавляем слушатель действий для Alpha.
        jbtnAlpha.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent ae) {
                jlab.setText("Alpha was pressed.");
              // jlab.setText("Нажата кнопка Alpha.");
            }
        });

        // Добавляем слушатель действий для Beta.
        jbtnBeta.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                jlab.setText("Beta was pressed.");
              // jlab.setText("Нажата кнопка Beta.");
            }
        });

        // Добавляем кнопки в панель содержимого.
        jfrm.add(jbtnAlpha);
        jfrm.add(jbtnBeta);

        // Создаем текстовую метку.
        jlab = new JLabel("Press a button.");
     // jlab = new JLabel("Нажмите кнопку.");

        // Добавляем метку в панель содержимого.
        jfrm.add(jlab);

        // Отображаем фрейм.
        jfrm.setVisible(true);
    }

    public static void main(String args[]) {
        // Создаем фрейм в потоке диспетчеризации событий.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new EventDemo();
            }
        });
    }
}
```

Для начала обратите внимание на то, что программа теперь импортирует пакеты `java.awt` и `java.awt.event`. Пакет `java.awt` нужен потому, что в нем содержится класс `FlowLayout`, поддерживающий стандартный диспетчер компоновки потоков, который используется для размещения компонентов во фрейме. (Диспетчеры компоновки рассматривались в главе 25.) Пакет `java.awt.event` необходим потому, что он определяет интерфейс `ActionListener` и класс `ActionEvent`.

Конструктор класса EventDemo начинает работу с создания контейнера класса JFrame по имени jfrm. Затем он устанавливает диспетчер компоновки класса FlowLayout для панели содержимого контейнера jfrm. Вспомните, что по умолчанию панель содержимого использует диспетчер компоновки класса BorderLayout. Однако для данного примера удобнее применить именно класс FlowLayout. Обратите внимание на то, что диспетчер класса FlowLayout назначается с помощью следующего оператора.

```
jfrm.setLayout(new FlowLayout());
```

Как уже упоминалось, раньше приходилось явным образом вызывать метод getContentPane(), чтобы задать диспетчер компоновки для окна содержимого. После выхода комплекта JDK 5 этого делать не нужно.

После определения размеров и стандартной операции при закрытии метод EventDemo() создает две кнопки, как показано ниже.

```
JButton jbtnAlpha = new JButton("Alpha");
JButton jbtnBeta = new JButton("Beta");
```

Первая кнопка будет содержать текст "Alpha", а вторая — "Beta". Кнопки библиотеки Swing являются экземплярами класса JButton. Класс JButton предлагает несколько конструкторов. Одним из используемых здесь конструкторов является следующий.

```
JButton(String сообщение)
```

Параметр *сообщение* определяет строку, которая будет отображаться внутри кнопки.

При щелчке на кнопке происходит событие класса ActionEvent. Таким образом, класс JButton предлагает метод addActionListener(), который используется для добавления слушателя событий. (Класс JButton предлагает также метод removeActionListener() для удаления слушателя, однако он в этой программе не используется.) Как было сказано в главе 23, интерфейс ActionListener определяет только один метод — actionPerformed(). Чтобы вам было легче его вспомнить, приведем его еще раз.

```
void actionPerformed(ActionEvent ae)
```

Этот метод вызывается в результате щелчка на кнопке. Другими словами, это обработчик события, который вызывается в случае события при щелчке на кнопке.

После этого показанный ниже код добавляет слушатели для событий действий с кнопками.

```
// Добавляем слушатель событий действий для кнопки Alpha.
jbtnAlpha.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alpha was pressed.");
    }
});

// Добавляем слушатель событий действий для кнопки Beta.
jbtnBeta.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta was pressed.");
    }
});
```

Здесь анонимные внутренние классы используются для того, чтобы предоставить обработчики событий двум кнопкам. Всякий раз при щелчке на кнопке строка, отображенная в метке jlab, изменяется в зависимости от того, на какой кнопке был произведен щелчок.

Затем кнопки добавляются в панель содержимого.

```
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);
```

И наконец, в панель содержимого добавляется метка `jlab`, и окно становится видимым. Когда вы запустите программу, то при каждом щелчке на кнопке в метке будет отображаться сообщение, указывающее, на какой из кнопок был щелчок.

И еще одно замечание: не забывайте о том, что все обработчики событий наподобие метода `actionPerformed()` вызываются в потоке диспетчеризации событий. Таким образом, обработчик событий должен быстро дать результат, чтобы не допустить замедления работы приложения. Если вашему приложению нужно сделать что-то такое, для чего потребуется много времени и что будет расцениваться как событие, то оно должно использовать отдельный поток.

Создание аплета Swing

Еще одним типом программы, которая обычно использует классы библиотеки Swing, является аплет. Аплеты, созданные на основе библиотеки Swing, подобны аплетам, созданным на основе библиотеки AWT, но у них есть одно существенное отличие — аплет Swing расширяет класс `JApplet`, а не класс `Applet`. Таким образом, класс `JApplet` включает все функциональные возможности класса `Applet` и добавляет поддержку библиотеки Swing. Класс `JApplet` является контейнером библиотеки Swing верхнего уровня; это значит, что он не является наследником класса `JComponent`. Так как класс `JApplet` является контейнером верхнего уровня, он включает различные панели, описанные ранее. Это означает, что все компоненты добавляются в панель содержимого класса `JApplet` точно так же, как и компоненты, добавляемые в панель содержимого `JFrame`.

Аплеты Swing используют те же методы обеспечения жизненного цикла, которые были описаны в главе 22, — `init()`, `start()`, `stop()` и `destroy()`. Естественно, вам нужно переопределять только те методы, которые будут нужны вашему аплету. Процесс рисования в библиотеке Swing осуществляется иначе, чем в библиотеке AWT, и аплет Swing обычно не переопределяет метод `paint()`. (О рисовании с использованием библиотеки Swing поговорим далее в этой главе.)

Еще один нюанс: все взаимодействия с компонентами в библиотеке Swing должны производиться в потоке диспетчеризации событий, о чем было сказано в предыдущем разделе. Это относится ко всем программам Swing.

Ниже представлен пример аплета Swing. Он снабжен теми же функциями, что и предыдущее приложение, но только является аплетом. На рис. 30.3 показан аплет, выполняемый в приложении `appleteviewer`.

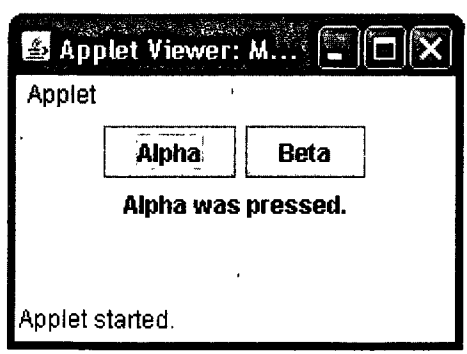


Рис. 30.3. Результат работы аплета Swing

```
// Простой аплет, основанный на библиотеке Swing

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/*
Этот код HTML можно использовать для запуска аплета:

<applet code="MySwingApplet" width=220 height=90>
</applet>
*/

public class MySwingApplet extends JApplet {
    JButton jbtnAlpha;
    JButton jbtnBeta;

    JLabel jlab;

    // Инициализация аплета.
    public void init() {
        try {
            SwingUtilities.invokeAndWait(new Runnable () {
                public void run() {
                    makeGUI(); // инициализация GUI
                }
            });
        } catch(Exception exc) {
            System.out.println("Can't create because of "+ exc);
         // System.out.println("Невозможно создать из-за "+ exc);
        }
    }

    // Этому аплету не нужно переопределять методы start(), stop()
    // или destroy().

    // Настройка и инициализация GUI.
    private void makeGUI() {

        // Настройка аплета для использования компоновки потоков.
        setLayout(new FlowLayout());

        // Создание двух кнопок.
        jbtnAlpha = new JButton("Alpha");
        jbtnBeta = new JButton("Beta");

        // Добавление слушателя событий действия для кнопки Alpha.
        jbtnAlpha.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent le) {
                jlab.setText("Alpha was pressed.");
             // jlab.setText("Нажата кнопка Alpha.");
            }
        });

        // Добавление слушателя событий действия для кнопки Beta.
        jbtnBeta.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent le) {
                jlab.setText("Beta was pressed.");
             // jlab.setText("Нажата кнопка Beta.");
            }
        });
```

```
        // Добавление кнопок в панель содержимого.
        add(jbtnAlpha);
        add(jbtnBeta);

        // Создание текстовой метки.
        jlab = new JLabel("Press a button.");
      // jlab = new JLabel("Нажмите кнопку.");

        // Добавление метки в панель содержимого.
        add(jlab);
    }
}
```

Необходимо сделать пару важных замечаний касательно аплетов. Во-первых, класс `MySwingApplet` расширяет класс `JApplet`. Как мы уже говорили, все аплеты, основанные на библиотеке Swing, расширяют класс `JApplet`, а не класс `Applet`. Во-вторых, метод `init()` инициализирует компоненты библиотеки Swing в потоке диспетчеризации событий, устанавливая вызов метода `makeGUI()`. Обратите внимание на то, что для этого используется метод `invokeAndWait()`, а не `invokeLater()`. Аплеты должны использовать метод `invokeAndWait()` потому, что метод `init()` не должен возвращать результат до тех пор, пока не будет выполнен весь процесс инициализации. По сути, метод `start()` нельзя вызывать прежде, чем закончится инициализация; это значит, что нужно полностью создать GUI.

Внутри метода `makeGUI()` создаются две кнопки и метка, а к кнопкам добавляются слушатели событий действия. Наконец, компоненты добавляются в панель содержимого. Несмотря на то что пример является довольно простым, этот подход нужно применять при создании любого GUI библиотеки Swing, который будет использован аплетом.

Рисование с использованием библиотеки Swing

Несмотря на то что компоненты библиотеки Swing очень функциональны, вы не ограничены только их использованием, поскольку библиотека Swing позволяет также выводить информацию непосредственно в область отображения фрейма, панели или одного из компонентов библиотеки Swing, такого как метка класса `JLabel`. Хотя во многих случаях использования библиотеки Swing не производится рисование прямо на поверхности компонента, это можно делать в приложениях, где подобное необходимо. Чтобы вывести данные прямо на поверхность компонента, нужно использовать один или несколько методов рисования, определенных в библиотеке AWT, вроде `drawLine()` или `drawRect()`. Таким образом, большинство технологий и методов, описанных в главе 24, можно применять и к библиотеке Swing. С другой стороны, между ними есть очень важные отличия, поэтому обо всем этом мы и поговорим подробно в текущем разделе.

Основы рисования

Подход к рисованию с использованием библиотеки Swing базируется на оригинальном механизме, построенном на основе библиотеки AWT, однако библиотека Swing позволяет более качественно управлять этим процессом. Прежде чем приступить к изучению специфики рисования в библиотеке Swing, будет полезно пересмотреть механизм рисования в библиотеке AWT.

Класс Component библиотеки AWT предлагает метод paint(), который используется для рисования выходных данных прямо на поверхности компонента. Как правило, метод paint() не вызывается программой. (В действительности, вызывать этот метод написанной вами программой придется в очень редких случаях.) Вместо этого метод paint() вызывается средой выполнения в процессе визуализации компонента. Такая ситуация может возникнуть в силу ряда причин. Например, окно, в котором отображается компонент, может быть перекрыто другим окном, а затем вновь появиться на экране. Или же окно может быть свернуто, а затем восстановлено. Метод paint() вызывается также в тех случаях, когда программа начинает свою работу. При написании кода, основанного на библиотеке AWT, приложение будет переопределять метод paint(), когда ему будет необходимо вывести данные прямо на поверхности компонента.

Поскольку класс JComponent унаследован от класса Component, все облегченные компоненты библиотеки Swing получают метод paint(). Однако вам *не* придется переопределять его, чтобы выводить информацию непосредственно на поверхности компонента. Дело в том, что при рисовании с использованием библиотеки Swing применяется более изощренный подход, который включает три различных метода: paintComponent(), paintBorder() и paintChildren(). Эти методы рисуют заданную часть компонента и делят процесс рисования на три разных логических действия. В облегченном компоненте исходный метод paint() библиотеки AWT просто выполняет вызовы этих методов в показанном только что порядке.

Чтобы нарисовать поверхность компонента библиотеки Swing, вы должны будете создать подкласс компонента, а затем переопределить его метод paintComponent(). Этот метод отвечает за прорисовку внутренней части компонента. Как правило, остальные два метода рисования вы переопределять не будете. Переопределяя метод paintComponent(), вы сначала должны вызвать метод super.paintComponent(), чтобы задействовать часть суперкласса процесса рисования. (Этого делать не нужно лишь в том случае, если вы вручную управляете способом отображения компонента.) После этого можно вывести данные, которые вы хотите отобразить. Ниже показан метод paintComponent().

```
protected void paintComponent(Graphics g)
```

В параметре *g* указывается графическое содержимое выходных данных.

Чтобы программно нарисовать компонент, нужно вызвать метод repaint(). Он работает в библиотеке Swing точно так же, как в библиотеке AWT. Метод repaint() определен в классе Component. Если его вызвать, система вызывает метод paint() сразу же, как только представляется для этого возможность. Поскольку процесс рисования отнимает довольно много времени, этот механизм позволяет системе времени выполнения мгновенно задерживать рисование до тех пор, пока, например, не завершится выполнение другой задачи, имеющей более высокий приоритет. Естественно, в библиотеке Swing вызов метода paint() приводит к вызову метода paintComponent(). Таким образом, чтобы вывести данные на поверхность компонента, ваша программа будет хранить эти данные до тех пор, пока не будет вызван метод paintComponent(). Внутри переопределенного метода paintComponent() вы будете рисовать хранимые данные.

Вычисление области рисования

Во время рисования на поверхности компонента нужно тщательно ограничить область рисования выходных данных, находящуюся внутри границ компонента. Хотя библиотека Swing автоматически отсекает любые выходные данные, выходящие за

границы компонента, может получиться так, что рисование будет выполняться прямо на границе, которая затем будет заменена при перерисовке. Чтобы не допустить этого, следует вычислить *область рисования* в пределах компонента. Эта область определяется так: берется текущий размер компонента и из него вычитается пространство, занятое его границами. Таким образом, прежде чем нарисовать компонент, следует сначала узнать ширину границы, а затем уже подгонять область рисования.

Чтобы узнать ширину границы, вызовите метод `getInsets()`.

```
Insets getInsets()
```

Этот метод определен в классе `Container` и переопределяется в классе `JComponent`. Он возвращает объект класса `Insets`, содержащий размеры границ. Значения можно получить с помощью следующих полей.

```
int top;
int bottom;
int left;
int right;
```

Впоследствии эти значения применяются для вычисления области рисования с учетом ширины и высоты компонента. Ширину и высоту компонента можно узнать, обратившись к методам `getWidth()` и `getHeight()`.

```
int getWidth()
int getHeight()
```

Вычитая значения границ, можно получить ширину и высоту области, в которой будет выполняться рисование.

Пример рисования

Сейчас рассмотрим программу, в которой будут реализованы рассмотренные методы. Она создает класс `PaintPanel`, расширяющий класс `JPanel`. Программа использует объект данного класса для отображения линий, конечные точки которых создаются случайным образом. Результат выполнения программы показан на рис. 30.4.

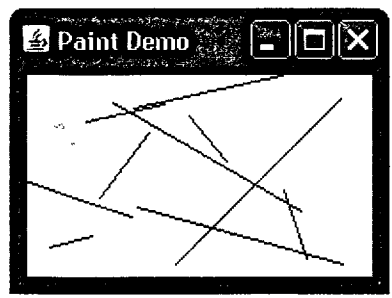


Рис. 30.4. Результат выполнения программы `PaintPanel`

```
// Рисование линий в панели.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

// Этот класс расширяет класс JPanel. Он переопределяет
// метод paintComponent(), чтобы выводить
// в панели случайные линии.
```

```
class PaintPanel extends JPanel {
    Insets ins; // хранит размеры внутренней части панели

    Random rand; // используется для создания случайных чисел

    // Создаем панель.
    PaintPanel() {

        // Помещаем рамку вокруг панели (создаем ее границы).
        setBorder(
            BorderFactory.createLineBorder(Color.RED, 5));

            rand = new Random();
        }

        // Переопределяем метод paintComponent().
        protected void paintComponent(Graphics g) {
            // Первым всегда вызывается метод суперкласса.
            super.paintComponent(g);
            int x, y, x2, y2;

            // Получаем высоту и ширину компонента.
            int height = getHeight();
            int width = getWidth();

            // Получаем размеры внутренней части.
            ins = getInsets();

            // Рисуем десять линий, конечные точки которых
            // создаются случайным образом.
            for(int i=0; i < 10; i++) {
                // Получаем 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);
                // Рисуем линию.
                g.drawLine(x, y, x2, y2);
            }
        }
    }

    // Иллюстрация рисования непосредственно в панели.
    class PaintDemo {
        JLabel jlab;
        PaintPanel pp;
        PaintDemo() {

            // Создаем новый контейнер JFrame.
            JFrame jfrm = new JFrame("Paint Demo");

            // Присваиваем фрейму исходные размеры.
            jfrm.setSize(200, 150);

            // Прекращаем работу программы, если пользователь
            // закрывает приложение.
            jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

            // Создаем панель, которую будем рисовать.
            pp = new PaintPanel();

            // Добавляем панель на панель содержимого. Так как
```

```
        // используется компоновка BorderLayout, размеры панели будут
        // подбираться таким образом, чтобы она заняла центральную
        // часть области.
        jfrm.add(pp);

        // Отображаем фрейм.
        jfrm.setVisible(true);
    }

    public static void main(String args[]) {
        // Создаем фрейм в потоке диспетчеризации событий.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
            new PaintDemo();
            }
        });
    }
}
```

А теперь обсудим программу. Класс `PaintPanel` расширяет класс `JPanel`, который является одним из облегченных контейнеров библиотеки Swing, т.е. мы получаем компонент, который можно добавлять в панель содержимого `JFrame`. Для обработки процесса рисования класс `PaintPanel` переопределяет метод `paintComponent()`. Это позволяет классу `PaintPanel` рисовать прямо на поверхности компонента. Размеры панели не определены, поскольку программа по умолчанию использует компоновку класса `BorderLayout`, а панель добавляется в центр области. В результате этого панель получает такие размеры, которые позволяют ей уместиться в центре. При изменении размеров окна будут соответствующим образом подгоняться и размеры панели.

Обратите внимание на то, что конструктор также определяет красную рамку (границу) толщиной в 5 пикселей. Для этого используется метод `setBorder()`, показанный ниже.

```
void setBorder(Border рамка)
```

Интерфейс `Border` библиотеки Swing инкапсулирует рамку. Рамку можно получить, вызвав один из методов, определенных в классе `BorderFactory`. В этой программе применяется один из таких методов — `createLineBorder()`. Он создает простую линейную рамку.

```
static Border createLineBorder(Color clr, int ширина)
```

Здесь параметр *clr* определяет цвет рамки, а *ширина* — ее ширину в пикселях.

Внутри переопределения метода `paintComponent()` следует обратить внимание на то, что он сначала вызывает метод `super.paintComponent()`. Как уже было сказано, это необходимо для обеспечения надлежащего рисования. Затем вычисляется ширина и высоты панели, а также внутренней части. Эти значения используются для того, чтобы рисуемые линии находились внутри области рисования панели. Область рисования — это общая ширина и высота компонента минус ширина рамки. Вычисления реализованы так, чтобы можно было оперировать различными размерами области рисования и границ. Чтобы убедиться в этом, попробуйте изменить размеры окна. Линии по-прежнему будут рисоваться внутри границ панели.

Класс `PaintDemo` создает класс `PaintPanel`, а затем добавляет панель в панель содержимого. Во время первого отображения приложения вызывается переопределенный метод `paintComponent()` и рисуются линии. Каждый раз, когда будете изменять размеры окна или сворачивать и восстанавливать его, будет рисоваться новый набор линий. Во всех классах линии будут находиться внутри заданной области рисования.

ГЛАВА 31

Дополнительные сведения о библиотеке Swing

В предыдущей главе были рассмотрены некоторые базовые концепции библиотеки Swing и показана общая форма приложения и аплета, основанных на библиотеке Swing. В этой главе продолжим обсуждение библиотеки Swing, предоставляя описание ряда ее компонентов, таких как кнопки, флажки, деревья и таблицы. Компоненты библиотеки Swing обладают богатыми функциональными возможностями и предлагают широкие возможности по их настройке. К сожалению, в этой книге невозможно описать все особенности и атрибуты компонентов, поэтому рассмотрим лишь некоторые из них.

Классы компонентов библиотеки Swing, рассматриваемых в этой главе, показаны ниже в виде таблицы.

`JButton`	`JCheckBox`	`JComboBox`	`JLabel`
`JList`	`JRadioButton`	`JScrollPane`	`JTabbedPane`
`JTable`	`JTextField`	`JToggleButton`	`JTree`

Все эти компоненты являются облегченными; это означает, что все они происходят от класса `JComponent`.

В этой главе будут использоваться также класс `ButtonGroup`, который инкапсулирует набор взаимоисключающих кнопок библиотеки Swing, и класс `ImageIcon`, инкапсулирующий графическое изображение. Каждый из них определен в библиотеки Swing и включен в пакет `javax.swing`.

И еще один момент. Компоненты библиотеки Swing демонстрируются в аплетах, поскольку код аплета более компактный, чем код настольного приложения, а технологии к ним применяются одни и те же.

Классы `JLabel` и `ImageIcon`

Самым простым в использовании компонентом библиотеки Swing является класс `JLabel`. Мы рассматривали его в предыдущей главе, и вы уже знаете, что он создает метку. В этой главе поговорим о нем подробнее. Класс `JLabel` можно использовать для отображения текста и/или значка (пиктограммы). Этот компонент является пассивным в том смысле, что не реагирует на данные, вводимые пользователем. Класс `JLabel` определяет несколько конструкторов. Ниже показаны три из них.

```
JLabel(Icon значок)
JLabel(String строка)
JLabel(String строка, Icon значок, int выравнивание)
```

Здесь параметры *строка* и *значок* представляют текст и значок, которые будут использоваться для метки. Параметр *выравнивание* определяет выравнива-

ние текста и/или значка по горизонтали внутри метки. Он должен иметь одно из следующих значений: LEFT, RIGHT, CENTER, LEADING или TRAILING. Наряду с некоторыми другими константами, используемыми в классах библиотеки Swing, эти константы определены в интерфейсе SwingConstants.

Обратите внимание на то, что значки определяются с помощью объектов, имеющих тип Icon, который представляет собой интерфейс, определенный в библиотеке Swing. Получить значок проще всего можно при помощи класса ImageIcon. Класс ImageIcon реализует Icon и инкапсулирует изображение. Таким образом, объект класса ImageIcon можно передать с помощью параметра Icon конструктора класса JLabel. Предоставить изображение можно несколькими способами, включая чтение изображения из файла или его загрузку из места, определенного с помощью адреса URL. Ниже показан конструктор класса ImageIcon, используемый в примере этого раздела.

```
ImageIcon(String имяфайла)
```

Он получает изображение из файла, имя которого указано в параметре *имяфайла*.

Значок и текст, связанные с меткой, можно получить с помощью следующих методов.

```
Icon getIcon()
String getText()
```

Значок и текст, связанные с меткой, можно установить с применением таких методов.

```
void setIcon(Icon значок)
void setText(String строка)
```

Здесь *значок* и *строка* представляют значок и текст соответственно. Таким образом, с помощью метода setText() можно изменить текст внутри метки во время работы программы.

На примере следующего аплета показано, как создается и отображается метка, содержащая значок и строку. Аплет начинается с создания объекта класса ImageIcon из файла france.gif, который отображает государственный флаг Франции. Это изображение используется в качестве второго параметра конструктора класса JLabel. Первый и последний параметры конструктора класса JLabel представляют текст метки и его выравнивание. В конце аплета метка добавляется в панель содержимого.

```
// Пример JLabel и ImageIcon.
import java.awt.*;
import javax.swing.*;
/*
<applet code="JLabelDemo" width=250 height=150>
</applet>
*/

public class JLabelDemo extends JApplet {

    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
```

```
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
          // System.out.println("Невозможно создать из-за " + exc);
        }
    }

    private void makeGUI() {
        // Создание значка.
        ImageIcon ii = new ImageIcon("france.gif");
        // Создание метки.
        JLabel jl = new JLabel("France", ii, JLabel.CENTER);
        // Добавление метки в панель содержимого.
        add(jl);
    }
}
```

На рис. 31.1 показана полученная метка.

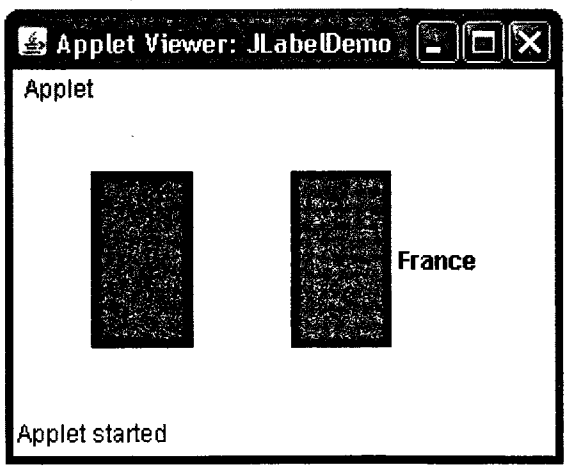


Рис. 31.1. Окно аплета `JLabelDemo`

Класс `JTextField`

Класс `JTextField` — это простейший текстовый компонент библиотеки Swing. Более того, это, пожалуй, наиболее широко используемый текстовый компонент. Класс `JTextField` позволяет редактировать одну строку текста. Он происходит от класса `JTextComponent`, который наделяет функциональными возможностями текстовые компоненты библиотеки Swing. Для своей модели класс `JTextField` использует интерфейс `Document`.

Ниже показаны три конструктора класса `JTextField`.

```
JTextField(int цвета)
JTextField(String строка, int цвета)
JTextField(String строка)
```

Здесь *строка* — это первоначально представляемая строка, а *цвета* — количество столбцов в текстовом поле. Если строка не будет задана, то текстовое поле первоначально будет пустым. Если не будет задано количество столбцов, то размер текстового поля будет выбран так, чтобы оно могло уместиться в указанной строке.

Класс `JTextField` создает извещение о событии в ответ на действия пользователя. Например, событие класса `ActionEvent` происходит при нажатии пользователем клавиши <Enter>. Событие `CaretEvent` происходит при каждом изменении позиции каретки (т.е. курсора). (Событие `CaretEvent` определено в пакете `javax.swing.event`.) Произойти могут и другие события. Как правило, вашей программе не нужно будет обрабатывать эти события. Наоборот, вы будете просто получать строку, находящуюся в данный момент в текстовом поле. Для ее получения необходимо вызвать метод `getText()`.

Ниже показан пример применения класса `JTextField`. В этом аплете создается объект класса `JTextField` и добавляется в панель содержимого. Когда пользователь нажимает клавишу <Enter>, происходит событие действия. В результате этого события отображается текст в окне состояния.

```
// Демонстрация применения JTextField.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/

public class JTextFieldDemo extends JApplet {
    JTextField jtf;
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI( );
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
         // System.out.println("Невозможно создать из-за " + exc);
        }
    }

    private void makeGUI() {
        // Изменение компоновки потока.
        setLayout(new FlowLayout());
        // Добавление текста в панель содержимого.
        jtf = new JTextField(15);
        add(jtf);
        jtf.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                // Отображаем текст, когда пользователь нажимает Enter.
                showStatus(jtf.getText());
            }
        });
    }
}
```

На рис. 31.2 показан пример полученного текстового поля.

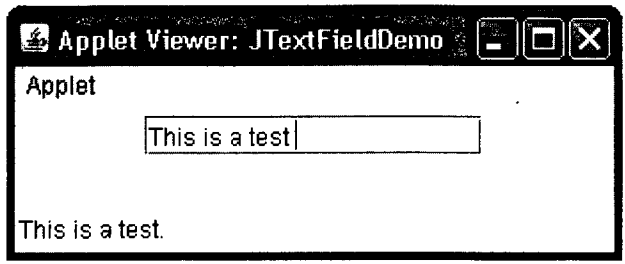


Рис. 31.2. Окно аплета `JTextFieldDemo`

Кнопки библиотеки Swing

В библиотеке Swing определено четыре класса кнопок — `JButton`, `JToggleButton`, `JCheckBox` и `JRadioButton`. Все они являются подклассами класса `AbstractButton`, который расширяет класс `JComponent`. Таким образом, у кнопок есть общие характерные черты.

Класс `AbstractButton` содержит множество методов, позволяющих управлять поведением кнопок. Например, вы можете определить различные значки, которые будут отображаться на месте кнопки, когда она будет отключена, нажата или выбрана. Другой значок можно использовать в качестве значка-наката (rollover), который будет отображаться при наведении указателя мыши на кнопку. Ниже показаны методы, с помощью которых можно задавать эти значки.

```
void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
void setRolloverIcon(Icon ri)
```

Параметры *di*, *pi*, *si* и *ri* определяют значки, используемые для различных состояний. Текст, связанный с кнопкой, можно прочитать и записать при помощи следующих методов.

```
String getText()
void setText(String строка)
```

Здесь *строка* — это текст, который будет связан с кнопкой.

Модель, используемая всеми кнопками, определена интерфейсом `ButtonModel`. Кнопка извещает о событии действия, когда ее нажимает пользователь. Возможны также и другие события. Сейчас поговорим о каждом конкретном классе кнопки.

Класс `JButton`

Класс `JButton` определяет функциональные возможности экранной кнопки. В предыдущей главе вы уже могли с ним вкратце познакомиться. Он позволяет связывать с кнопкой на экране значок, строку или оба этих элемента вместе. Ниже показаны три его конструктора.

```
JButton(Icon значок)
JButton(String строка)
JButton(String строка, Icon значок)
```

Здесь *значок* и *строка* представляют значок и строку, которые используются для кнопки. Когда пользователь нажимает кнопку (щелкает на ней), происходит событие класса `ActionEvent`. С помощью объекта класса `ActionEvent`, переданного методу `actionPerformed()` зарегистрированного слушателя

ActionListener, вы можете получить *командную строку действия*, связанную с кнопкой. По умолчанию эта строка будет отображаться внутри кнопки. Однако команду действия можно задать, вызвав метод setActionCommand() в отношении кнопки. Получить команду действия можно, вызвав метод getActionCommand() для объекта события. Он объявляется следующим образом.

```
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 был показан пример текстовой кнопки. В следующем примере демонстрируется кнопка со значком. В нем отображаются четыре кнопки и одна метка. Каждая кнопка отображает значок, представляющий флаг государства. Когда пользователь щелкает на кнопке, в метке появляется название государства.

```
// Пример использования кнопки JButton со значком.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=450>
</applet>
*/
public class JButtonDemo extends JApplet
implements ActionListener {
    JLabel jlab;
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
         // System.out.println("Невозможно создать из-за " + exc);
        }
    }

    private void makeGUI() {
        // Изменение компоновки потока.
        setLayout(new FlowLayout());
        // Добавление кнопок в панель содержимого.
        ImageIcon france = new ImageIcon("france.gif");
        JButton jb = new JButton(france);
        jb.setActionCommand("France");
        jb.addActionListener(this);
        add(jb);
        ImageIcon germany = new ImageIcon("germany.gif");
        jb = new JButton(germany);
        jb.setActionCommand("Germany");
        jb.addActionListener(this);
        add(jb);

        ImageIcon italy = new ImageIcon("italy.gif");
        jb = new JButton(italy);
        jb.setActionCommand("Italy");
        jb.addActionListener(this);
        add(jb);
```

```
        ImageIcon japan = new ImageIcon("japan.gif");
        jb = new JButton(japan);
        jb.setActionCommand("Japan");
        jb.addActionListener(this);
        add(jb);

        // Создаем и добавляем метку в панель содержимого.
        jlab = new JLabel("Choose a Flag");
        add(jlab);
    }

    // Обработка событий кнопки.
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("You selected " + ae.getActionCommand());
     // jlab.setText("Выбрана " + ae.getActionCommand());
    }
}
```

На рис. 31.3 показан результат выполнения этого аплета.

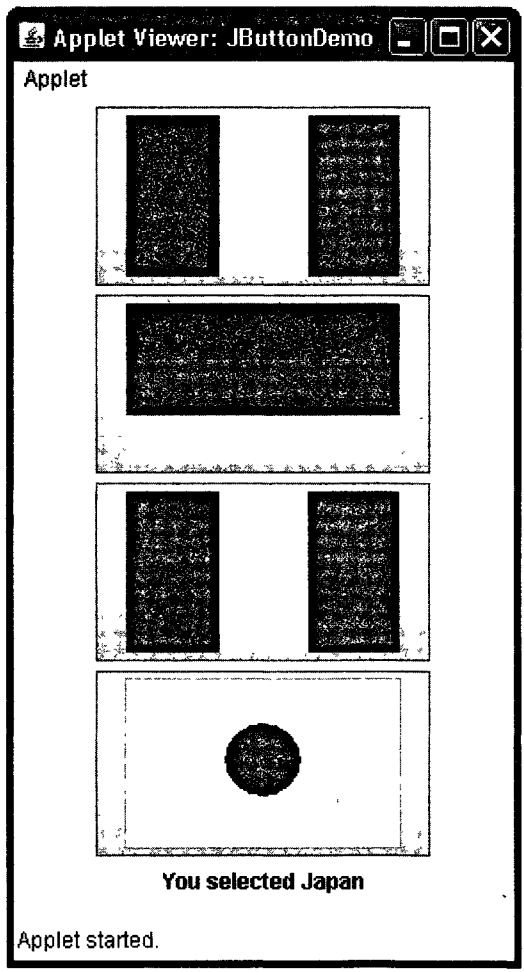


Рис. 31.3. Окно аплета `JButtonDemo`

Класс JToggleButton

Полезной разновидностью кнопки является *тумблер* (toggle button). Тумблер выглядит подобно обычной кнопке, но действует по-другому, так как у него может быть два состояния: нажатое и отжатое. То есть, когда вы нажимаете тумблер, он остается нажатым, а не отжимается подобно обычной кнопке. Если после этого нажать тумблер еще раз, он будет отжат. Таким образом, каждый раз, когда пользователь нажимает тумблер, последний принимает одно из двух возможных состояний.

Тумблеры являются объектами класса `JToggleButton`. Класс `JToggleButton` реализует класс `AbstractButton`. Кроме создания стандартных тумблеров, `JToggkeButton` является суперклассом двух других компонентов библиотеки Swing, которые также представляют элементы управления, имеющие два состояния. Это классы `JCheckBox` и `JRadioButton`. Таким образом, класс `JToggleButton` определяет базовые функции компонентов, обладающих двумя состояниями.

Класс `JToggleButton` определяет несколько конструкторов. Один из них, используемый в примере этого раздела, выглядит так.

```
JToggleButton(String строка)
```

Он создает тумблер, содержащий текст, заданный с помощью параметра *строка*. Стандартное состояние тумблера — отжатое. Остальные конструкторы позволяют создавать тумблеры, содержащие изображения или изображения и текст.

Класс `JToggleButton` использует модель, определенную во вложенном классе `JToggleButton.ToggleButtonModel`. *Как правило, вам не придется взаимодействовать непосредственно с моделью, чтобы использовать стандартный тумблер.*

Как и класс `JButton`, класс `JToggleButton` создает извещение о событии действия каждый раз, когда пользователь нажимает тумблер. Однако, в отличие от класса `JButton`, класс `JToggleButton` также создает извещение о событии элемента. Это событие используется компонентами, которые поддерживают принцип выбора. Если тумблеркласса `JToggleButton` нажат, он является выбранным. Если пользователь отжимает его, выбор отменяется.

Для обработки событий элемента нужно реализовать интерфейс `ItemListener`. Из главы 23 вы должны помнить, что при каждом создании извещения о событии элемента оно передается методу `itemStateChanged()`, определенному в интерфейсе `ItemListener`. В методе `itemStateChanged()` метод `getItem()` может быть вызван в объекте класса `ItemEvent`, чтобы получить ссылку на экземпляр класса `JToggleButton`, создавший извещение о событии. Этот метод показан ниже.

```
Object getItem()
```

Он возвращает ссылку на кнопку. Вам нужно будет привести эту ссылку к классу `JToggleButton`.

Самый простой способ определить состояние тумблера предусматривает вызов метода `isSelected()` (он является наследником класса `AbstractButton`) для кнопки, создавшей извещение о событии. Этот метод выглядит следующим образом.

```
boolean isSelected()
```

Он возвращает значение `true`, если кнопка была выбрана, в противном случае он возвращает значение `false`.

Ниже показан пример, в котором используется тумблер. Обратите внимание на слушателя событий. Он просто вызывает метод `isSelected()`, чтобы определить состояние кнопки.

```
// Демонстрация применения JToggleButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JToggleButtonDemo" width=200 height=80>
</applet>
*/

public class JToggleButtonDemo extends JApplet {

    JLabel jlab;
    JToggleButton jtbn;

    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Невозможно создать из-за " + exc);
        }
    }

    private void makeGUI() {

        // Изменение компоновки потока.
        setLayout(new FlowLayout());

        // Создаем метку.
        jlab = new JLabel("Button is off.");
        // jlab = new JLabel("Кнопка отжата.");

        // Создаем тумблер.
        jtbn = new JToggleButton("On/Off");

        // Добавляем слушатель событий для тумблера.
        jtbn.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent ie) {
                if(jtbn.isSelected())
                    jlab.setText("Button is on.");
                 // jlab.setText("Кнопка нажата.");
                else
                    jlab.setText("Button is off.");
                 // jlab.setText("Кнопка отжата.");
            }
        });

        // Добавляем тумблер и метку в панель содержимого.
        add(jtbn);
        add(jlab);
    }
}
```

Результат выполнения этого аплета показан на рис. 31.4.

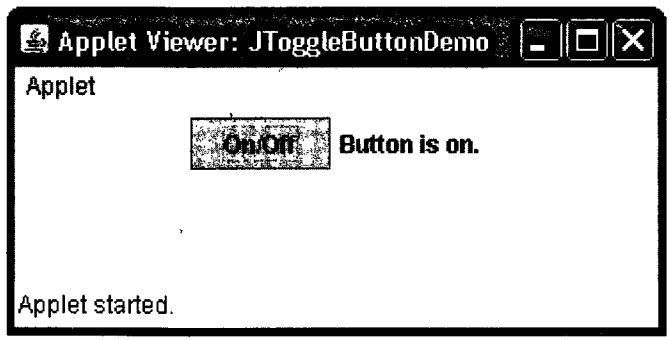


Рис. 31.4. Окно аплета `JToggleButtonDemo`

Флажки

Класс `JCheckBox` определяет функции флажка. Его суперклассом является класс `JToggleButton`, который обеспечивает поддержку кнопок с двумя состояниями (о них мы только что говорили). Класс `JCheckBox` определяет несколько конструкторов. Один из них выглядит следующим образом.

```
JCheckBox(String строка)
```

Этот конструктор создает флажок с текстом, определенным с помощью параметра *строка* в качестве метки. Остальные конструкторы позволяют определить исходное состояние выбора кнопки и задать значок.

Если пользователь устанавливает или сбрасывает флажок, создается извещение о событии класса `ItemEvent`. Вы можете получить ссылку на флажок класса `JCheckBox`, создавший извещение о событии, вызвав метод `getItem()` в объекте класса `ItemEvent`, который передан методу `itemStateChanged()`, определенному в интерфейсе `ItemListener`. Определить выбранное состояние проще всего, вызвав метод `isSelected()` в экземпляре класса `JCheckBox`.

В следующем аплете демонстрируется использование флажков. Аплет отображает четыре флажка и метку. Когда пользователь щелкает на флажке, происходит событие класса `ItemEvent`. В методе `itemStateChanged()` вызывается метод `getItem()` для получения ссылки на объект класса `JCheckBox`, создавший извещение о событии. После этого вызов метода `isSelected()` определяет состояние флажка – установлен или сброшен. Метод `getText()` получает текст для данного флажка и использует его для определения текста внутри метки.

```
// Демонстрация применения JCheckbox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=270 height=50>
</applet>
*/

public class JCheckBoxDemo extends JApplet
implements ItemListener {
    JLabel jlab;

    public void init() {
        try {
```

```
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );

        } catch (Exception exc) {
                System.out.println("Can't create because of " + exc);
             // System.out.println("Невозможно создать из-за " + exc);
        }
    }

    private void makeGUI() {

        // Изменение компоновки потока.
        setLayout(new FlowLayout());

        // Добавление флажков в панель содержимого.
        JCheckBox cb = new JCheckBox("C");
        cb.addItemListener(this);
        add(cb);

        cb = new JCheckBox("C++");
        cb.addItemListener(this);
        add(cb);

        cb = new JCheckBox("Java");
        cb.addItemListener(this);
        add(cb);

        cb = new JCheckBox("Perl");
        cb.addItemListener(this);
        add(cb);

        // Создание метки и добавление ее в панель содержимого.
        jlab = new JLabel("Select languages");
     // jlab = new JLabel("Выберите языки");
        add(jlab);
    }

    // Обработка событий флажка.
    public void itemStateChanged(ItemEvent ie) {
        JCheckBox cb = (JCheckBox)ie.getItem();

        if(cb.isSelected())
             jlab.setText(cb.getText() + " is selected");
        // jlab.setText(cb.getText() + " выбран");
        else
            jlab.setText(cb.getText() + " is cleared");
         // jlab.setText(cb.getText() + " сброшен");
    }
}
```

На рис. 31.5 показан результат выполнения аплета.

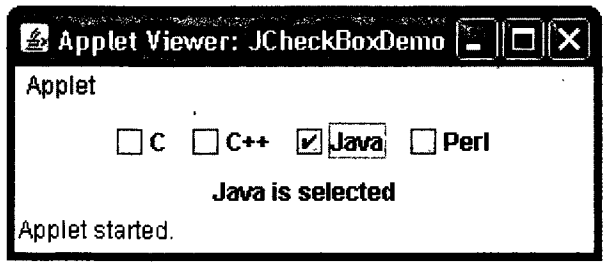


Рис. 31.5. Окно аплета `JCheckBoxDemo`

Переключатели

Переключатели представляют собой группы взаимоисключающих кнопок, в которых можно выбрать только одну. Они поддерживаются классом `JRadioButton`, который расширяет класс `JToggleButton`. Класс `JRadioButton` предлагает несколько конструкторов. Один из них, используемый в примере, показан ниже.

```
JRadioButton(String строка)
```

Здесь *строка* — это метка кнопки. Остальные конструкторы позволяют определить исходное состояние кнопки и задать значок.

Переключатели необходимо объединять в группу, в которой можно выбрать только одну из кнопок. Например, если пользователь выбирает переключатель, находящийся в группе, то выбранный ранее переключатель в этой группе автоматически отключается. Для создания группы кнопок предназначен класс `ButtonGroup`. Для этой цели вызывается его конструктор, используемый по умолчанию. После этого можно добавить в группу элементы с помощью следующего метода.

```
void add(AbstractButton ab)
```

Здесь *ab* представляет ссылку на кнопку, которую необходимо добавить в группу.

Класс `JRadioButton` создает извещение о событии действий, события элементов и события изменений каждый раз, когда меняется выбранная кнопка в переключателе. Как правило, обрабатывается событие действия, а это означает, что вы обычно будете реализовать интерфейс `ActionListener`. Вы должны помнить, что интерфейс `ActionListener` определяет только один метод — `actionPerformed()`. Узнать, какая кнопка была выбрана, в этом методе можно несколькими способами. Во-первых, можно проверить ее команду действия, вызвав метод `getActionCommand()`. По умолчанию команда действия — это то же, что и метка кнопки, однако вы можете задать вместо нее что-нибудь другое, вызвав метод `setActionCommand()` для переключателя. Во-вторых, можно вызвать метод `getSource()` в объекте класса `ActionEvent` и проверить ссылку относительно кнопок. Наконец, можно просто проверить каждый переключатель, чтобы узнать, какая кнопка в данный момент была выбрана, вызвав метод `isSelected()` для каждой кнопки. Помните, что каждый раз, когда происходит событие действия, оно означает, что выбранная кнопка была изменена и что была выбрана только одна кнопка.

В следующем аплете демонстрируется использование переключателей. В нем создаются и объединяются в группу три переключателя. Как уже было сказано, это нужно для того, чтобы они взаимно исключали друг друга. При выборе переключателя происходит событие действия, которое обрабатывается методом `actionPerformed()`. В этом обработчике метод `getActionCommand()` получает текст, который связан с переключателем, и использует его для определения текста в метке.

```
// Демонстрация применения JRadioButton
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet
implements ActionListener {
    JLabel jlab;
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
         // System.out.println("Невозможно создать из-за " + exc);
        }
    }

    private void makeGUI() {
        // Изменение компоновки потока.
        setLayout(new FlowLayout());
        // Создание переключателей и добавление их в панель содержимого.
        JRadioButton b1 = new JRadioButton("A");
        b1.addActionListener(this);
        add(b1);

        JRadioButton b2 = new JRadioButton("B");
        b2.addActionListener(this);
        add(b2);

        JRadioButton b3 = new JRadioButton("C");
        b3.addActionListener(this);
        add(b3);
        // Определение группы кнопок.
        ButtonGroup bg = new ButtonGroup();
        bg.add(b1);
        bg.add(b2);
        bg.add(b3);

        // Создание метки и добавление ее в панель содержимого.
        jlab = new JLabel("Select One");
     // jlab = new JLabel("Выберите один из переключателей");
        add(jlab);
    }
    // Обработка выбора кнопки.
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("You selected " + ae.getActionCommand());
     // jlab.setText("Выбран " + ae.getActionCommand());
    }
}
```

Результат выполнения этого аплета показан на рис. 31.6.

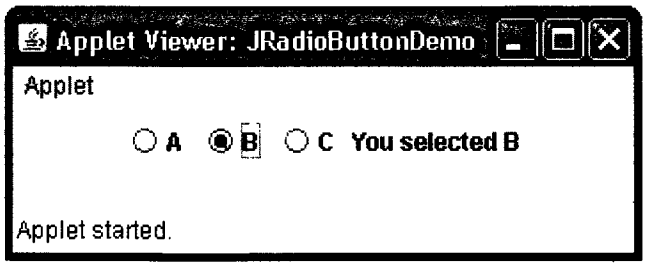


Рис. 31.6. Окно аплета `JRadioButtonDemo`

Класс JTabbedPane

Класс `JTabbedPane` определяет *панель с вкладками*. Он управляет набором компонентов, соединяя их с помощью вкладок. При выборе вкладки компонент, связанный с ней, появляется на переднем плане. Панели с вкладками очень популярны в современных пользовательских графических интерфейсах, и вы тоже будете часто их применять в своих программах. Несмотря на сложную природу панели с вкладками, создавать и использовать их очень просто.

Класс `JTabbedPane` определяет три конструктора. Мы будем использовать стандартный конструктор, который создает пустой элемент управления с вкладками, расположенными вдоль верхней части панели. Другие два конструктора позволяют определить месторасположение вкладок, которые можно расположить вдоль одной из четырех сторон. Класс `JTabbedPane` использует модель `SingleSelectionModel`.

Вкладки добавляются при помощи вызова метода `addTab()`. Ниже показана одна из его форм.

```
void addTab(String имя, Component компаратор)
```

Здесь *имя* — это имя вкладки, а *компаратор* — это компонент, который должна содержать вкладка. Обычно добавляется компонент `JPanel`, содержащий группу связанных между собой компонентов. Эта технология позволяет вкладке содержать набор компонентов.

Общая процедура использования панели с вкладками выглядит следующим образом.

1. Создается объект класса `JTabbedPane`.
2. Для добавления каждой вкладки нужно вызвать метод `addTab()`.
3. Панель с вкладками переносится в панель содержимого.

В следующем примере показан процесс создания панели с вкладками. Первая вкладка имеет заголовок Cities (Города) и содержит четыре кнопки. Каждая кнопка отображает название города. Вторая вкладка имеет заголовок Colors (Цвета) и содержит три флажка. Каждый флажок отображает название цвета. Третья вкладка имеет заголовок Flavors (Привкусы) и содержит один комбинированный список. С его помощью пользователь может выбрать один из трех привкусов.

```
// Демонстрация применения JTabbedPane.
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
```

```
*/
public class JTabbedPaneDemo extends JApplet {
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
         // System.out.println("Невозможно создать из-за " + exc);
        }
    }

    private void makeGUI() {
        JTabbedPane jtp = new JTabbedPane();
        jtp.addTab("Cities", new CitiesPanel());
        jtp.addTab("Colors", new ColorsPanel());
        jtp.addTab("Flavors", new FlavorsPanel());
        add(jtp);
    }
}

// Создаем панели, которые будут добавляться 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);
    }
}
```

На рис. 31.7 показаны окна этого аплета.

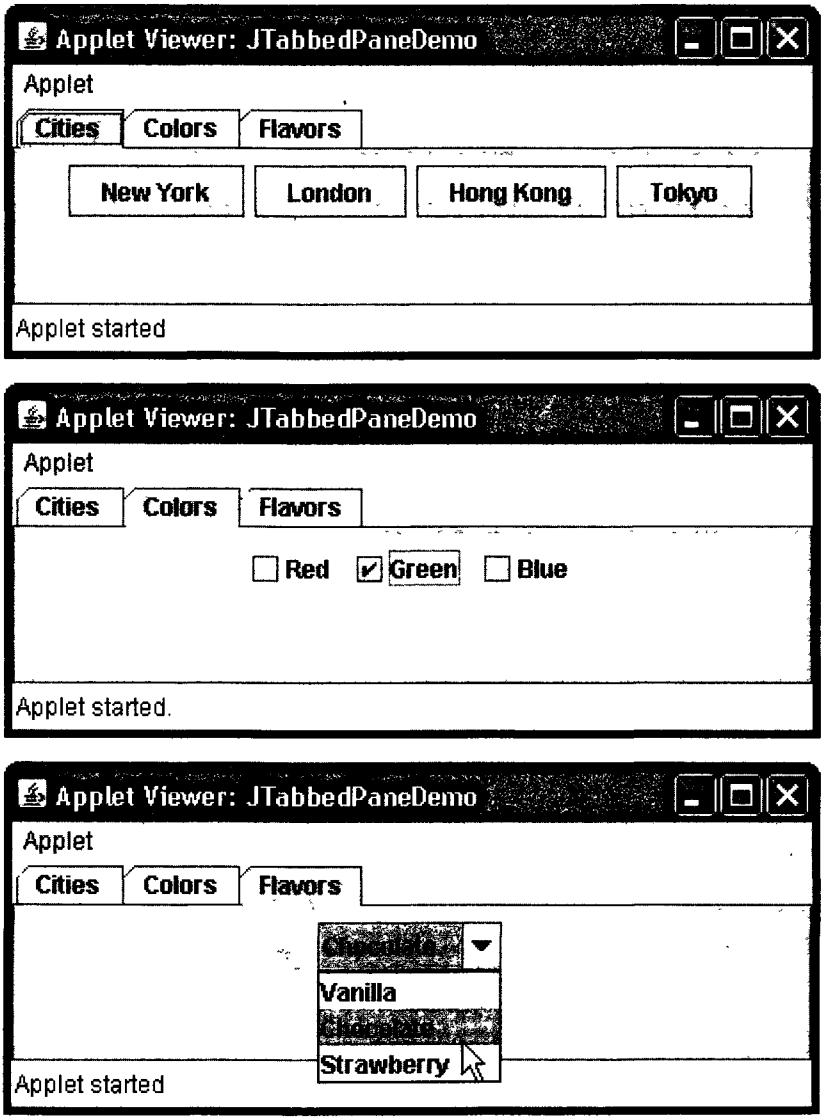


Рис. 31.7. Окна аплета `JTabbedPaneDemo`

Класс `JScrollPane`

Класс `JScrollPane` — это облегченный контейнер, который автоматически обрабатывает прокрутку другого компонента. Прокручиваемым компонентом может быть как отдельный компонент (например, таблица), так и группа компонентов, содержащихся внутри другого облегченного контейнера, такого как `JPanel`. В любом случае, если прокручиваемый объект больше области просмотра, к нему

автоматически добавляется горизонтальная и/или вертикальная полоса прокрутки, что позволяет прокручивать компонент в рамках панели. Поскольку класс `JScrollPane` автоматизирует процесс прокрутки, он обычно исключает необходимость в управлении отдельными полосами прокрутки.

Просматриваемая область панели с полосами прокрутки называется *окном просмотра*. Это — окно, в котором отображается прокручиваемый компонент. Таким образом, окно просмотра показывает видимую часть прокручиваемого компонента. Полосы прокрутки прокручивают компонент в области просмотра. По умолчанию класс `JScrollPane` динамически добавляет или удаляет полосу прокрутки по мере необходимости. Например, если компонент выше окна просмотра, добавляется вертикальная полоса прокрутки. Если компонент полностью умещается в окне просмотра, полосы прокрутки удаляются.

Класс `JScrollPane` определяет несколько конструкторов. Конструктор, используемый в этой главе, выглядит так.

`JScrollPane(Component` *`компаратор`*`)`

Прокручиваемый компонент задается с помощью параметра *компаратор*. Полосы прокрутки автоматически отображаются, если содержимое панели превышает размеры окна просмотра.

Чтобы использовать панель с полосами прокрутки, нужно выполнить следующие действия.

1. Создать компонент, который будет прокручиваться.
2. Создать экземпляр класса `JScrollPane`, передавая ему объект прокрутки.
3. Добавить панели с полосами прокрутки в панель содержимого.

На примере следующего аплета демонстрируется применение панели с полосами прокрутки. Сначала в нем создается объект `JPanel`, после чего в этот объект добавляется 400 кнопок, сгруппированных в 20 столбцов. Затем эта панель добавляется в панель с полосами прокрутки, а последняя добавляется в панель содержимого. Поскольку панель больше окна просмотра, то автоматически появляются вертикальная и горизонтальная полосы прокрутки. Полосы прокрутки можно использовать для прокрутки кнопок в данном представлении.

```
// Демонстрация применения JScrollPane.
import java.awt.*;
import javax.swing.*;
/*
<applet code="JScrollPaneDemo" width=300 height=250>
</applet>
*/

public class JScrollPaneDemo extends JApplet {
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
          // System.out.println("Невозможно создать из-за " + exc);
        }
    }
```

```
    private void makeGUI() {
        // Добавление 400 кнопок в панель.
        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;
            }
        }
        // Создание панели с прокруткой.
        JScrollPane jsp = new JScrollPane(jp);

        // Добавление панели с прокруткой в панель содержимого.
        // Так как используется компоновка границ по умолчанию,
        // панель с полосами прокрутки добавляется в центр.
        add(jsp, BorderLayout.CENTER);
    }
}
```

На рис. 31.8 показан результат выполнения этого аплета.

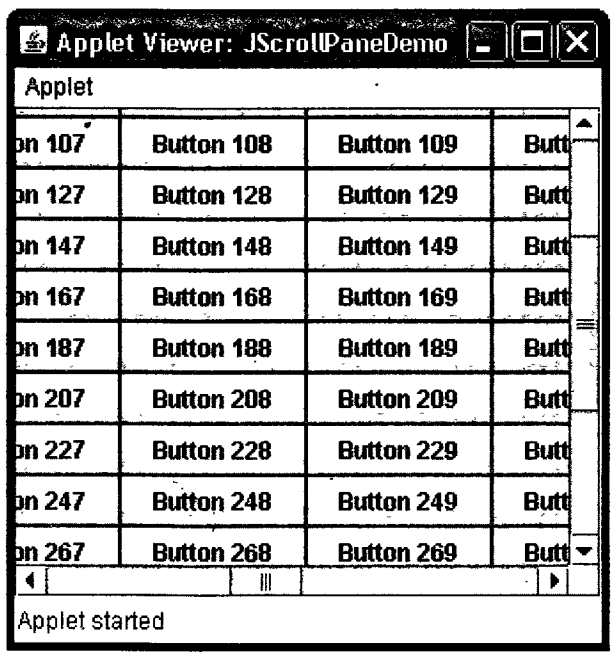


Рис. 31.8. Окно аплета `JScrollPaneDemo`

Класс `JList`

Базовым классом списков в библиотеке Swing является класс `JList`. Он поддерживает выбор одного или нескольких элементов из списка. Зачастую список состоит из строк, но можно создать список из любых объектов, которые только можно отобразить. Класс `JList` настолько часто применяется в Java, что вы уже должны были обязательно с ним встретиться.

Ранее элементы в объекте класса JList были представлены как ссылки на класс Object. Но с выпуском комплекта JDK 7 класс JList был сделан обобщением и теперь объявляется так.

```
class JList<E>
```

Здесь параметр *E* представляет тип элементов в списке.

Класс JList предлагает несколько конструкторов. Один из наиболее используемых конструкторов выглядит так.

```
JList(E[] элементы)
```

Этот конструктор создает список класса JList, содержащий элементы в массиве, определяемые с помощью параметра *элементы*.

Класс JList основан на двух моделях. Первая модель — интерфейс ListModel, который определяет, как осуществляется доступ к данным списка. Второй моделью является интерфейс ListSelectionModel, который определяет методы, позволяющие узнать, какой элемент (элементы) списка был выбран.

Хотя класс JList способен без проблем работать самостоятельно, обычно его экземпляр вкладывают в контейнер класса JScrollPane. Благодаря этому длинные списки автоматически прокручиваются, что упрощает проектирование графического интерфейса. Это позволит также упростить изменение количества записей в списке, не изменяя его размеры.

Класс JList создает извещение о событии ListSelectionEvent, когда пользователь выбирает элемент или изменяет выбор элемента. Это событие происходит также тогда, когда пользователь отменяет выбор элемента. Оно обрабатывается слушателем ListSelectionListener, который определяет только один метод — valueChanged().

```
void valueChanged(ListSelectionEvent le)
```

Здесь параметр *le* — это ссылка на объект, который создал извещение о событии.

Хотя класс события ListSelectionEvent тоже предлагает некоторые методы, вы обычно будете использовать сам объект класса JList, чтобы узнать, что произошло.

Событие ListSelectionEvent и слушатель ListSelectionListener определены в пакете javax.swing.event.

По умолчанию класс JList позволяет пользователю выбирать несколько диапазонов элементов внутри списка, однако вы можете изменить это поведение, вызвав метод setSelectionModel(), который определен в классе JList. Он показан ниже.

```
void setSelectionMode(int режим)
```

Здесь *режим* — это режим выбора. Он должен быть представлен одним из следующих значений, определенных интерфейсом ListSelectionModel.

```
SINGLE_SELECTION
SINGLE_INTERVAL_SELECTION
MULTIPLE_INTERVAL_SELECTION
```

По умолчанию используется последнее значение, которое позволяет пользователю выбирать множество диапазонов элементов внутри списка. Если будет задан выбор в единичном интервале (SINGLE_INTERVAL_SELECTION), пользователь сможет выбрать только один диапазон элементов. Если будет выбрано значение SINGLE_SELECTION, пользователь сможет выбрать только один элемент. Естественно, один элемент можно выбрать и в двух других режимах. Просто эти режимы позволяют также выбирать диапазон элементов.

Вы можете получить индекс первого выбранного элемента, который будет также являться индексом единственного выбранного элемента в режиме SINGLE_SELECTION, если вызовете метод getSelectedIndex(), показанный ниже.

```
int getSelectedIndex()
```

Индексация начинается с нуля. Поэтому если будет выбран первый элемент, метод вернет значение 0. Если не будет выбрано ни одного элемента, будет возвращено значение -1.

Вместо того чтобы получать индекс выбранного элемента, вы можете получить значение, связанное с выбранным элементом, вызвав метод getSelectedValue().

```
E getSelectedValue()
```

Метод возвращает ссылку на первое выбранное значение. Если не будет выбрано ни одного значения, он вернет значение null.

На примере следующего аплета демонстрируется использование простого списка класса JList, хранящего перечень городов. Каждый раз, когда пользователь выбирает город в списке, происходит событие ListSelectionEvent, обработкой которого занимается метод valueChanged(), определенный в слушателе ListSelectionListener. Он получает индекс выбранного элемента и отображает имя выбранного города в метке.

```
// Демонстрация применения JList.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

/*
<applet code="JListDemo" width=200 height=120>
</applet>
*/

public class JListDemo extends JApplet {
    JList<String> jlst;
    JLabel jlab;
    JScrollPane jscrlp;
    // Создаем массив городов.
    String Cities[] = { "New York", "Chicago", "Houston",
                        "Denver", "Los Angeles", "Seattle",
                        "London", "Paris", "New Delhi",
                        "Hong Kong", "Tokyo", "Sydney" };
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
         // System.out.println("Невозможно создать из-за " + exc);
        }
    }

    private void makeGUI() {
        // Изменение компоновки потока.
        setLayout(new FlowLayout());

        // Создаем список JList.
        jlst = new JList<String>(Cities);

        // Присваиваем режиму выбора значение SINGLE_SELECTION.
        jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

```
        // Добавляем список в панель с полосами прокрутки.
        jscrlp = new JScrollPane(jlst);

        // Задаем предпочтительные размеры панели с полосами прокрутки.
        jscrlp.setPreferredSize(new Dimension(120, 90));

        // Создаем метку, в которой будет отображаться выбранный город.
        jlab = new JLabel("Choose a City");
        // jlab = new JLabel("Выберите город");

        // Добавляем слушатель события выбора для списка.
        jlst.addListSelectionListener(new ListSelectionListener() {
            public void valueChanged(ListSelectionEvent le) {
                // Получаем индекс измененного элемента.
                int idx = jlst.getSelectedIndex();
                // Отображаем выбор, если элемент был выбран.
                if(idx != -1)
                    jlab.setText("Current selection: " + Cities[idx]);
                 // jlab.setText("Текущий выбор: " + Cities[idx]);
                else // В противном случае повторно предлагаем выбрать
                     // город.
                    jlab.setText("Choose a City");
                 // jlab.setText("Выберите город");
            }
        });
        // Добавляем список и метку в панель содержимого.
        add(jscrlp);
        add(jlab);
    }
}
```

Результат выполнения этого аплета показан на рис. 31.9.

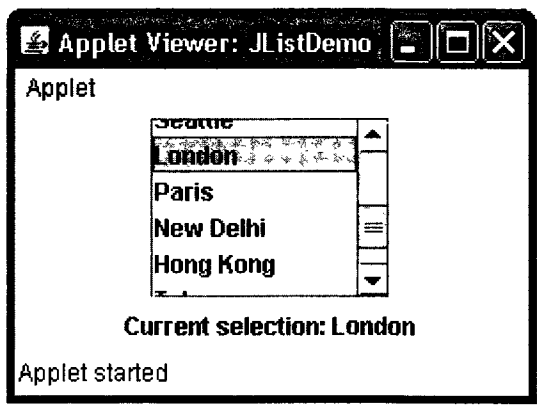


Рис. 31.9. Окно аплета `JListDemo`

Класс JComboBox

С помощью класса `JComboBox` библиотеки Swing определяется компонент, называемый *комбинированным списком* (комбинация текстового поля и раскрывающегося списка). Как правило, комбинированный список отображает одну запись, но он будет также отображать и раскрывающийся список, позволяющий выбирать другие элементы. Вы можете создать комбинированный список, который позволит вводить выбираемый элемент в текстовом поле.

Ранее элементы в объекте класса JComboBox были представлены как ссылки на класс Object. Но с выпуском JDK 7 класс JComboBox был сделан обобщением и теперь объявляется так.

```
class JComboBox<E>
```

Здесь параметр *E* представляет тип элементов в раскрывающемся списке.

Ниже показан конструктор класса JComboBox, используемый в этом примере.

```
JComboBox(E[] элементы)
```

Здесь *элементы* — это массив, инициализирующий комбинированный список. Кроме него, доступны также и другие конструкторы.

Класс JComboBox использует модель ComboBoxModel. Изменяемые комбинированные списки (списки, элементы которых могут изменяться) используют модель MutableComboBoxModel.

Кроме передачи массива элементов, которые должны быть отображены в раскрывающемся списке, элементы можно добавлять динамически в список при помощи метода addItem(), показанного ниже.

```
void addItem(E объект)
```

Здесь *объект* — это объект, который необходимо добавить в комбинированный список. Этот метод должен использоваться только в изменяемых комбинированных списках.

Класс JComboBox создает извещение о событии действия, когда пользователь выбирает элемент из списка. Класс JComboBox также создает извещение о событии элемента, когда изменяется состояние выбора, что происходит при выборе или отмене выбора элемента. Таким образом, при изменении выбора происходит два события: одно — для элемента, выбор которого был отменен, а другое — для выбранного элемента. Нередко оказывается достаточно простого прослушивания событий действия, однако использовать можно оба типа событий.

Узнать, какой элемент списка был выбран, можно с помощью метода getSelectedItem().

```
Object getSelectedItem()
```

Вам нужно будет привести возвращенное значение к типу объекта, хранящегося в списке. На примере следующего аплета показано использование комбинированного списка. В этом окне содержатся элементы "France", "Germany", "Italy" и "Japan". Если пользователь выберет государство, произойдет обновление метки, чтобы в ней был отображен флаг данного государства. Обратите внимание на то, насколько мало кода нужно для того, чтобы использовать этот мощный компонент.

```
// Демонстрация применения JComboBox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JComboBoxDemo" width=300 height=100>
</applet>
*/

public class JComboBoxDemo extends JApplet {
    JLabel jlab;
    ImageIcon france, germany, italy, japan;
    JComboBox<String> jcb;
    String flags[] = { "France", "Germany", "Italy", "Japan" };
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
```

```
                public void run() {
                    makeGUI();
                }
            }
        );
    } catch (Exception exc) {
        System.out.println("Can't create because of " + exc);
      // System.out.println("Невозможно создать из-за " + exc);
    }
}

private void makeGUI() {

    // Изменение компоновки потока.
    setLayout(new FlowLayout());

    // Создание экземпляра комбинированного списка
    // и добавление его в панель содержимого.
    jcb = new JComboBox<String>(flags);
    add(jcb);

    // Обработка выбранных элементов.
    jcb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            String s = (String) jcb.getSelectedItem();
            jlab.setIcon(new ImageIcon(s + ".gif"));
        }
    });

    // Создание метки и добавление ее в панель содержимого.
    jlab = new JLabel(new ImageIcon("france.gif"));
    add(jlab);
}
}
```

На рис. 31.10 показан результат выполнения аплета.

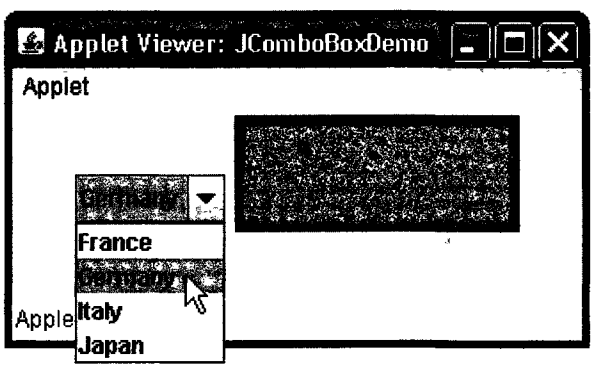


Рис. 31.10. Окно аплета `JComboBoxDemo`

Деревья

Дерево — это компонент, представляющий иерархический вид данных. В данном представлении пользователь может развертывать или свертывать отдельные узлы. В библиотеке Swing деревья реализуются при помощи класса `JTree`. Ниже показаны некоторые из его конструкторов.

```
JTree(Object объект[ ])
JTree(Vector<?> v)
JTree(TreeNode tn)
```

В первом случае дерево создается из элементов массива *объект*, во втором случае — из элементов вектора (параметр *v*). В третьем случае дерево определяется корневым узлом, который, в свою очередь, определяется соответствующим параметром (*tn*).

Хотя класс `JTree` входит в состав пакета `javax.swing`, он поддерживает классы и интерфейсы, которые определены в пакете `javax.swing.tree`. Это объясняется тем, что количество классов и интерфейсов, необходимых для поддержки класса `JTree`, слишком большое.

Класс `JTree` базируется на двух моделях — `TreeModel` и `TreeSelectionModel`. Этот класс поддерживает разнообразные события, однако применительно к деревьям можно выделить три из них — `TreeExpansionEvent`, `TreeSelectionEvent` и `TreeModelEvent`. Событие `TreeExpansionEvent` происходит при развертывании или свертывании узла, событие `TreeSelectionEvent` — при выборе пользователем или отмене выбора узла в дереве, а событие `TreeModelEvent` — при изменении данных или структуры дерева. За этими событиями следят слушатели `TreeExpansionListener`, `TreeSelectionListener` и `TreeModelListener` соответственно. Классы событий дерева и интерфейсы слушателей определены в пакете `javax.swing.event`.

В примере программы этого раздела обрабатывается событие `TreeSelectionEvent`. Чтобы прослушать это событие, нужно реализовать слушатель `TreeSelectionListener`. Он определяет только один метод, `valueChanged()`, который получает объект класса `TreeSelectionEvent`. Вы можете получить путь к выбранному объекту, обратившись к методу `getPath()`.

```
TreePath getPath()
```

Метод возвращает объект класса `TreePath`, который описывает путь к измененному узлу. Класс `TreePath` инкапсулирует информацию о пути к определенному узлу в дереве. Он предлагает несколько методов и конструкторов. В этой книге используется только метод `toString()`. Он возвращает строку, описывающую путь.

Интерфейс `TreeNode` объявляет методы, которые получают информацию об узле дерева. Например, можно получить ссылку на родительский узел или перечень узлов-потомков. Интерфейс `MutableTreeNode` расширяет интерфейс `TreeNode`. Он объявляет методы, которые могут вставлять и удалять узлы-потомки или изменять родительский узел.

Класс `DefaultMutableTreeNode` реализует интерфейс `MutableTreeNode`. Он представляет узел в дереве. Ниже показан один из его конструкторов.

```
DefaultMutableTreeNode(Object объект)
```

Здесь *объект* — это объект, который необходимо заключить в данном узле дерева. Новый узел дерева не имеет родителя или потомка.

Чтобы создать иерархию из трех узлов, можно использовать метод `add()` класса `DefaultMutableTreeNode`. Ниже показана его сигнатура.

```
void add(MutableTreeNode потомок)
```

Здесь *потомок* — это изменяющийся узел дерева, который необходимо добавить в качестве потомка текущего узла.

Класс `JTree` сам по себе не предоставляет никаких возможностей для прокрутки. Поэтому экземпляр класса `JTree` обычно помещается внутрь контейнера класса `JScrollPane`. Таким образом, большое дерево можно прокрутить в окне просмотра с меньшими размерами.

Ниже перечислены действия, которые необходимо выполнить, чтобы использовать дерево в аплете.

1. Создайте экземпляр класса `JTree`.
2. Создайте экземпляр класса `JScrollPane` и определите дерево в качестве объекта прокрутки.
3. Добавьте дерево в панель с полосами прокрутки.
4. Добавьте панель с полосами прокрутки в панель содержимого.

В следующем примере показано, как создается дерево и обрабатывается выбор элементов. Программа создает объект класса `DefaultMutableTreeNode` с заголовком Options (Параметры). Он является верхним узлом в иерархии дерева. Затем создаются дополнительные узлы дерева и вызывается метод `add()` для присоединения этих узлов к дереву. Ссылку на верхний узел дерева обеспечивает параметр конструктора класса `JTree`. После этого дерево указывается в качестве параметра конструктора класса `JScrollPane`. Эта панель с полосами прокрутки добавляется в аплет. Затем создается метка и добавляется в панель содержимого. Эта метка отображает выбор в дереве. Чтобы получать извещения о событиях при выборе, регистрируется слушатель `TreeSelectionListener`. Внутри метода `valueChanged()` мы получаем и отображаем путь к текущему месту выбора.

```
// Демонстрация применения JTree.
import java.awt.*;
import javax.swing.event.*;
import javax.swing.*;
import javax.swing.tree.*;
/*
<applet code="JTreeDemo" width=400 height=200>
</applet>
*/

public class JTreeDemo extends JApplet {
    JTree tree;
    JLabel jlab;
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
         // System.out.println("Невозможно создать из-за " + exc);
        }
    }

    private void makeGUI() {

        // Создаем верхний узел дерева.
        DefaultMutableTreeNode top =
                                new DefaultMutableTreeNode("Options");

        // Создаем 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);
```

```
        // Создаем 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);

        // Создаем дерево.
        tree = new JTree(top);

        // Добавляем дерево в панель прокрутки.
        JScrollPane jsp = new JScrollPane(tree);

        // Добавляем панель с полосами прокрутки в панель содержимого.
        add(jsp);

        // Добавляем метку в панель содержимого.
        jlab = new JLabel();
        add(jlab, BorderLayout.SOUTH);

        // Обработка событий выбора в дереве.
        tree.addTreeSelectionListener(new TreeSelectionListener() {
            public void valueChanged(TreeSelectionEvent tse) {
                jlab.setText("Selection is " + tse.getPath());
             // jlab.setText("Выбрано " + tse.getPath());
            }
        });
    }
}
```

На рис. 31.11 показан результат выполнения этого аплета.

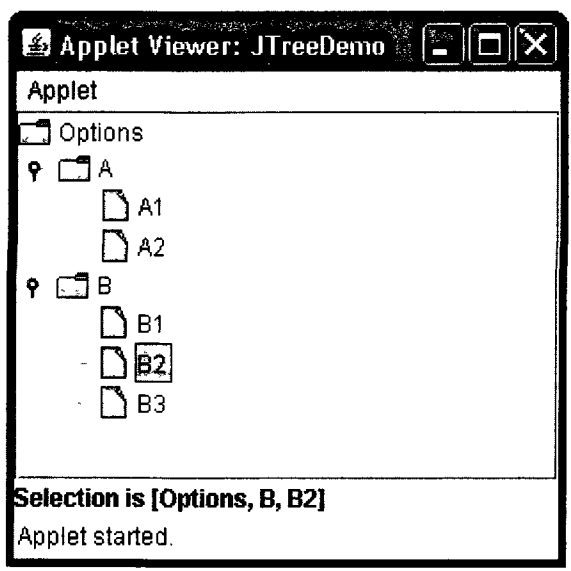


Рис. 31.11. Окно аплета `JTreeDemo`

Строка, представленная в текстовом поле, определяет путь из верхнего узла дерева к выбранному узлу.

Класс JTable

Класс JTable — это компонент, отображающий данные в виде строк и столбцов. Чтобы изменить размеры столбцов, вы можете перетаскивать их границы с помощью мыши. Кроме того, весь столбец можно перетащить в другую позицию. В зависимости от конфигурации, можно выбрать строку, столбец или ячейку в таблице, а также изменить данные в ячейке. Класс JTable является сложным компонентом, предлагающим гораздо больше параметров и функций, чем мы можем рассмотреть здесь. (Наверное, это наиболее сложный компонент библиотеки Swing.) Однако в своей стандартной конфигурации класс JTable предлагает легкие в использовании функции — особенно если вы просто хотите использовать таблицу для представления данных в табличном формате. Краткий обзор, представленный здесь, поможет вам понять, какими возможностями обладает этот компонент.

Как и класс JTree, класс JTable обладает многими классами и интерфейсами, связанными с ним. Все они находятся в пакете javax.swing.table.

В своей основе класс JTable является очень простым. Он состоит из одного или нескольких столбцов с информацией. Вверху каждого столбца расположен заголовок. Кроме описания данных в столбце, заголовок также предлагает механизм, при помощи которого пользователь может изменять размеры столбца или его местонахождение в таблице. Класс JTable не предлагает никаких возможностей прокрутки, поэтому вы обычно помещаете его в контейнер класса JScrollPane.

Класс JTable предлагает несколько конструкторов. Один из них выглядит следующим образом.

JTable(Object *данные*[ ][ ], Object *заголСтолбцов*[ ])

Здесь *данные* — это двухмерный массив информации, которую нужно представить, а *заголСтолбцов* — одномерный массив, содержащий заголовки столбцов. Класс JTable основан на трех моделях. Первой является модель таблицы, которая определена интерфейсом TableModel. Эта модель определяет все, что связано с отображением данных в двухмерном формате. Вторая модель — модель столбца таблицы, представленная с помощью TableColumnModel. Класс JTable определяет столбцы, а модель TableColumnModel — характеристики столбца. Эти две модели реализованы в пакете javax.swing.table. Третья модель определяет способ выбора элементов; она определяется с применением интерфейса ListSelectionModel (мы уже рассматривали ее при обсуждении класса JList).

В классе JTable может происходить несколько различных событий. Двумя наиболее фундаментальными событиями являются ListSelectionEvent и TableModelEvent. Событие ListSelectionEvent происходит при выборе пользователем чего-нибудь в таблице. По умолчанию класс JTable позволяет выбрать одну или несколько полных строк, однако вы можете изменить это поведение, чтобы позволить пользователю выбирать один или несколько столбцов или одну или несколько отдельных ячеек. Событие TableModelEvent происходит при изменении данных таблицы каким-либо образом. Обработка этих событий требует чуть больше затрат, чем в ранее описанных компонентах, и ее мы не сможем рассмотреть в этой книге. Но если вам просто понадобится использовать класс JTable для отображения данных (как в следующем примере), то никакие события обрабатывать не придется.

Для создания простой таблицы класса JTable, отображающей данные, выполните следующие действия.

1. Создайте экземпляр класса `JTable`.
2. Создайте экземпляр класса `JScrollPane`, определяя таблицу в качестве объекта прокрутки.
3. Добавьте таблицу в панель с полосами прокрутки.
4. Добавьте панель с полосами прокрутки в панель содержимого.

Ниже показан пример создания и использования простой таблицы. В этом аплете сначала создается одномерный массив строк `colHeads` для заголовков столбцов. Двухмерный массив строк `data` создается для данных ячеек таблицы. Каждый элемент в массиве является массивом из трех строк. Эти массивы передаются конструктору класса `JTable`. Таблица добавляется в панель с полосами прокрутки, после чего последняя добавляется в панель содержимого. Таблица отображает данные в массиве `data`. Конфигурация таблицы, используемая по умолчанию, позволяет также редактировать содержимое ячеек. Все изменения будут отражены в массиве `data`.

```
// Демонстрация применения JTable.
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTableDemo" width=400 height=200>
</applet>
*/

public class JTableDemo extends JApplet {

    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );

        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
          // System.out.println("Невозможно создать из-за " + exc);
        }
    }

    private void makeGUI() {

        // Инициализируем заголовки столбцов.
        String[] colHeads = { "Name", "Extension", "ID#" };

        // Инициализируем 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" }
        };

        // Создаем таблицу.
        JTable table = new JTable(data, colHeads);

        // Добавляем таблицу в панель с полосами прокрутки.
        JScrollPane jsp = new JScrollPane(table);

        // Добавляем панель с полосами прокрутки в панель содержимого.
        add(jsp);
    }
}
```

На рис. 31.12 показан результат выполнения этого аплета.

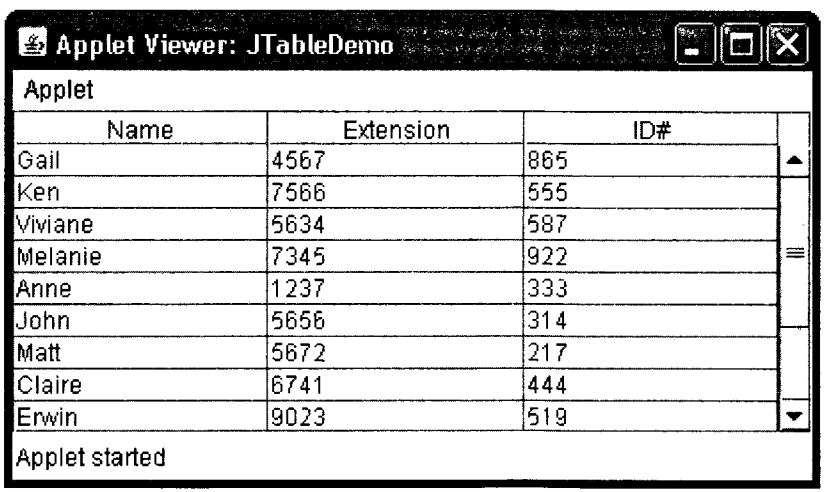


Рис. 31.12. Окно аплета `JTableDemo`

Продолжайте изучать библиотеку Swing

Библиотека Swing определяет большой набор инструментов пользовательского графического интерфейса. В ней заключены многие возможности, которые вам придется изучать самостоятельно. Например, библиотека Swing предлагает панели инструментов, контекстные подсказки и панели с индикаторами хода работ. Также библиотека Swing предлагает полную подсистему меню. Подключаемые внешние виды, которые тоже предлагает эта библиотека, позволяют получать разные варианты внешнего вида и поведения элемента. Вы можете определять собственные модели различных компонентов, а при работе с таблицами и деревьями изменять способ редактирования и визуализации данных в ячейках. Лучший способ изучения возможностей библиотеки Swing — постоянно экспериментировать с ними.

ГЛАВА 32 Сервлеты

В этой главе речь пойдет о сервлетах. *Сервлетами* (servlet) называются небольшие программы, которые выполняются на стороне сервера веб-соединения. Аплеты динамически расширяют функциональные возможности веб-браузера, а сервлеты – веб-сервера. Тема сервлетов довольно обширна и ее невозможно рассмотреть полностью в рамках одной главы. Поэтому мы сосредоточимся на рассмотрении концепций, интерфейсов и классов, а также проанализируем некоторые примеры.

Предварительные сведения

Чтобы разобраться с преимуществами сервлетов, следует иметь общее представление о том, как веб-браузеры и сервлеты работают сообща для предоставления содержимого пользователю. Рассмотрим запрос статической веб-страницы. Пользователь вводит в окне браузера адрес URL (Uniform Resource Locator – унифицированный указатель информационного ресурса). Браузер создает запрос HTTP к соответствующему веб-серверу. Веб-сервер устанавливает соответствие между запросом и конкретным файлом. Этот файл возвращается браузеру в виде ответа HTTP. Заголовок HTTP в ответе указывает тип содержимого. Для этого используется набор стандартов MIME (Multipurpose Internet Mail Extensions – многоцелевые расширения электронной почты). Например, обычный текст в формате ASCII имеет тип MIME `text/plain`. Исходный код HTML (Hypertext Markup Language – язык разметки гипертекста) имеет тип MIME `text/html`.

Теперь рассмотрим динамическое содержимое. Предположим, что магазин, работающий в интерактивном режиме, использует базу данных для хранения информации о своей бизнес-деятельности. База данных может включать элементы для регистрации продаж, прайс-листов, наличия товара, счетов и т.п. Руководство магазина решило сделать так, чтобы эта информация была доступна покупателям через веб-страницы. Содержимое этих веб-страниц должно создаваться динамически, чтобы отражать самую последнюю информацию в базе данных.

На ранних этапах существования системы веб-сервер мог динамически формировать страницу, создавая отдельный процесс для обработки каждого запроса клиента. Чтобы получить необходимую информацию, процесс мог открывать соединения с одной или несколькими базами данных. Связь с сервером осуществлялась при помощи интерфейса CGI (Common Gateway Interface – общий шлюзовой интерфейс). Интерфейс CGI позволял отдельным процессам считывать данные из запроса HTTP и записывать их в ответ HTTP. Для написания программ CGI применялись самые разные языки программирования. В их число входили языки C, C++ и Perl.

Однако у интерфейса CGI имелись серьезные проблемы, связанные с производительностью. Этот интерфейс был дорогим в плане потребления ресурсов про-

цессора и памяти, необходимых для создания отдельного процесса для каждого запроса клиента. Он был также дорогим в плане открытия и закрытия соединений с базой данных для каждого запроса клиента. Помимо всего этого, работа программ CGI зависела от конкретной платформы. Потому были предложены другие технологии, к числу которых относятся и сервлеты.

По сравнению с интерфейсом CGI, сервлеты обладают некоторыми преимуществами. Во-первых, их производительность заметно выше. Сервлеты выполняются внутри адресного пространства веб-сервера. Чтобы выполнить обработку каждого запроса клиента, не обязательно создавать отдельный процесс. Во-вторых, работа сервлетов не зависит от платформы, поскольку все они пишутся на языке Java. В-третьих, диспетчер безопасности Java на сервере реализует серию ограничений для защиты ресурсов на компьютере-сервере. И наконец, сервлету доступны абсолютно все функциональные возможности библиотек классов Java. Сервлет может работать с аплетами, базой данных и другим программным обеспечением при помощи сокетов и механизмов RMI, которые уже рассматривались в этой книге ранее.

Жизненный цикл сервлета

Жизненный цикл сервлета определяют три основных метода — `init()`, `service()` и `destroy()`. Они реализуются каждым сервлетом и вызываются сервером в определенное время. Сейчас рассмотрим обычный пользовательский сценарий, который поможет понять, когда происходит вызов этих методов.

Во-первых, предположим, что пользователь ввел в окне браузера адрес URL. На основании этого адреса браузер создает запрос HTTP, посылаемый соответствующему серверу.

Во-вторых, этот запрос HTTP принимает веб-сервер. Сервер находит соответствие между запросом и конкретным сервлетом. Сервлет динамически загружается в адресное пространство сервера.

В-третьих, сервер вызывает метод `init()` сервлета. Этот метод вызывается только тогда, когда сервлет впервые загружается в память компьютера. Сервлету можно передавать параметры инициализации, поэтому он может конфигурировать себя самостоятельно.

В-четвертых, сервер вызывает метод `service()` сервлета. Этот метод вызывается для обработки запроса HTTP. Вы увидите, что сервлет может считывать данные, содержащиеся в запросе HTTP. Он может также сформулировать ответ HTTP клиенту.

Сервлет остается в адресном пространстве и является доступным для обработки любых других запросов HTTP, полученных от клиентов. Метод `service()` вызывается для каждого запроса HTTP.

И наконец, сервер может принять решение выгрузить сервлет из памяти. Для принятия этого решения каждый сервер использует различные алгоритмы. Для освобождения ресурсов, таких как индексы файлов, выделенных для сервлета, сервер вызывает метод `destroy()`. Важные данные могут быть сохранены на постоянном носителе. Память, отведенная для сервлета и его объектов, впоследствии может быть утилизирована в процессе сбора "мусора".

Возможности разработки сервлетов

Для создания сервлетов вам понадобится доступ к контейнеру или серверу сервлетов. Наиболее популярными из них являются сервер сервлетов Glassfish и контейнер сервлетов Tomcat. Сервер Glassfish от Oracle предоставляется комплектом

SDK Java EE. Он поддерживается интегрированной средой разработки NetBeans. Контейнер Tomcat — это продукт реализации с открытым исходным кодом, поддерживаемый организацией Apache Software Foundation. Он также может использоваться интегрированной средой разработки NetBeans. И контейнер Tomcat, и сервер Glassfish могут также использоваться с другими интегрированными средами разработки, такими как Eclipse. Примеры и описания в этой главе используют контейнер Tomcat по причинам, которые скоро станут очевидны.

Хотя интегрированные среды разработки, такие как NetBeans и Eclipse, очень полезны и могут упростить создание сервлетов, они не используются в этой главе. Способы, которыми вы разрабатываете и развертываете сервлеты, зависят от конкретной интегрированной среды разработки, и в этой книге просто невозможно рассмотреть их все. Кроме того, многие читатели будут использовать инструментальные средства командной строки, а не интегрированную среду разработки. Поэтому, если вы используете интегрированную среду, за информацией о разработке и развертывании сервлетов следует обратиться к ее инструкциям. Поэтому инструкции, приведенные здесь и в других местах этой главы, подразумевают, что используются только инструментальные средства командной строки. Таким образом, они подойдут почти для любого читателя.

Контейнер Tomcat используется в этой главе, поскольку, по мнению автора, так относительно проще запускать примеры сервлетов, используя только инструменты командной строки и текстовый редактор. Это также вполне доступно в различных средах программирования. Кроме того, поскольку используются только инструментальные средства командной строки, вам не придется загружать и устанавливать интегрированную среду разработки только для того, чтобы экспериментировать с сервлетами. Не забывайте, однако, что даже при разработке в среде, которая использует сервер Glassfish, представленные здесь концепции вполне применимы. Лишь механизм подготовки сервлета к проверке будет немного отличаться.

Помните! Инструкции по разработке и развертыванию сервлетов в этой главе подразумевают использование только контейнера Tomcat и инструментов командной строки. Если вы используете интегрированную среду разработки и другой контейнер или сервер сервлетов, обратитесь к документации для своей среды.

Использование контейнера Tomcat

В состав системы контейнера Tomcat входят библиотеки классов, документация и среда выполнения, для создания и проверки сервлетов. На момент написания этой книги было доступно для использования несколько версий контейнера Tomcat, включая 5.5.*x*, 6.0.*x* и 7.0.*x*. Все они будут работать с примерами в этой главе. Однако инструкции, подразумевающие использование версии 7.0.4, поддерживают спецификацию 3.0 сервлетов. Вы можете загрузить систему контейнера Tomcat по адресу `tomcat.apache.org`. Контейнеры Tomcat версий 6.0.*x* и 7.0.*x* поддерживаются как 32-, так и 64-разрядной операционной системой Windows. Вы должны выбрать версию, соответствующую вашей системе.

В примерах этой главы подразумевается использование 64-разрядной операционной системы Windows. Подразумевается также, что 64-битовая версия контейнера Tomcat 7.0.4 была распакована из корневого каталога непосредственно в заданное по умолчанию расположение.

```
C:\apache-tomcat-7.0.4-windows-x64\apache-tomcat-7.0.4\
```

Это расположение подразумевается в примерах данной книги. Если вы загрузите контейнер Tomcat в другое место (или используете другую его версию), то необ-

ходимо будет внести в примеры соответствующие изменения. Возможно, придется установить переменную среды окружения JAVA_HOME и указать в ней каталог, где установлен комплект разработки Java Development Kit.

На заметку! Все каталоги, представленные в этом разделе, подразумевают использование контейнера Tomcat 7.0.4. Если вы установите другую версию контейнера Tomcat, то придется откорректировать имена используемых каталогов и пути так, чтобы они соответствовали установленной версии.

После установки вы запускаете контейнер Tomcat, выбрав файл `startup.bat` в каталоге `bin`, расположенном непосредственно в каталоге `apache-tomcat-7.0.4`. Чтобы остановить контейнер Tomcat, запустите файл `shutdown.bat`, расположенный также в каталоге `bin`.

Классы и интерфейсы, необходимые для создания сервлетов, содержатся в файле `servlet-api.jar`, который находится в следующем каталоге.

```
C:\apache-tomcat-7.0.4-windows-x64\apache-tomcat-7.0.4\lib
```

Чтобы сделать файл `servlet-api.jar` доступным, обновите переменную среды CLASSPATH, чтобы она включала следующую строку.

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\common\lib\servlet-api.jar
```

В качестве альтернативного варианта этот файл классов можно определить во время компиляции сервлетов. Например, следующая команда компилирует первый пример сервлета.

```
javac HelloServlet.java -classpath "C:\apache-tomcat-7.0.4-windowsx64\
apache-tomcat-7.0.4\lib\servlet-api.jar"
```

После того как завершите компиляцию сервлета, нужно сделать так, чтобы контейнер Tomcat нашел его. Для этого необходимо поместить его в подкаталог `webapps` каталога Tomcat и ввести его имя в файле `web.xml`. Для упрощения всех этих действий в примерах данной главы применяется каталог и файл `web.xml`, который контейнер Tomcat использует для своих собственных образцов сервлетов. Таким образом, вам не нужно создавать никаких файлов или каталогов только для экспериментов с примерами этих сервлетов. Ниже приведена процедура, которую вам необходимо будет выполнить.

Сначала скопируйте файл класса сервлета в следующий каталог.

```
C:\apache-tomcat-7.0.4-windows-x64\apache-tomcat-7.0.4\webapps\examples\
WEB-INF\classes
```

Затем добавьте имя сервлета и отображение в файл `web.xml` в следующий каталог.

```
C:\apache-tomcat-7.0.4-windows-x64\apache-tomcat-7.0.4\webapps\examples\
WEB-INF
```

Например, если предположить, что первый пример будет называться `HelloServlet`, в раздел, описывающий сервлеты, вы добавите следующие строки.

```
<servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>HelloServlet</servlet-class>
</servlet>
```

Затем добавьте следующие строки в раздел, определяющий преобразования сервлетов.

```
<servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
```

```
    <url-pattern>/servlet/HelloServlet</url-pattern>
</servlet-mapping>
```

Выполните эти же действия для всех примеров.

Простой сервлет

Чтобы освоиться с ключевыми особенностями сервлетов, начнем с того, что создадим и проверим простой сервлет. Для этого необходимо выполнить перечисленные ниже основные действия.

1. Создайте и скомпилируйте исходный код сервлета. После этого скопируйте файл классов сервлета в соответствующий каталог и добавьте имя сервлета и преобразования в соответствующий файл `web.xml`.
2. Запустите контейнер Tomcat.
3. Запустите веб-браузер и запросите сервлет.

Теперь давайте рассмотрим каждое действие подробно.

Создание и компиляция исходного кода сервлета

Для начала создайте файл `HelloServlet.java`, который будет содержать следующую программу.

```
import java.io.*;
import javax.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();
    }
}
```

Рассмотрим эту программу. Для начала обратите внимание на то, что она импортирует пакет `javax.servlet`. Этот пакет содержит классы и интерфейсы, необходимые для создания сервлетов. К ним мы еще вернемся далее в этой главе. Программа определяет также класс `HelloServlet` как подкласс класса `GenericServlet`. Класс `GenericServlet` обладает функциональными возможностями, упрощающими создание сервлета. Например, он предлагает версии методов `init()` и `destroy()`, которые можно использовать "как есть". Вам нужно будет позаботиться только о методе `service()`.

Внутри класса `HelloServlet` осуществляется переопределение метода `service()` (унаследованного от класса `GenericServlet`). Этот метод обрабатывает запросы от клиента. Обратите внимание на то, что первым аргументом является объект интерфейса `ServletRequest`. Благодаря ему сервлет будет считывать данные, предоставленные в запросе клиента. Вторым аргументом является объект интерфейса `ServletResponce`. С его помощью сервлет сможет сформировать ответ клиенту.

При вызове метода `setContentType()` определяется тип MIME для ответа HTTP. В этой программе типом MIME является `text/html`. Он показывает, что браузеру необходимо интерпретировать содержимое в качестве исходного кода HTML.

Метод `getWriter()` получает объект класса `PrintWriter`. Все, что будет передаваться в поток, будет посылаться клиенту как часть ответа HTTP. Метод `println()` используется для записи некоторого простого исходного кода HTML в качестве ответа HTTP.

Скомпилируйте этот исходный код и поместите файл `HelloServlet.class` в соответствующий каталог контейнера Tomcat, руководствуясь описанием из предыдущего раздела. Кроме того, добавьте класс `HelloServlet` в файл `web.xml`, как было сказано ранее.

Запуск контейнера Tomcat

Запустите контейнер Tomcat, выполнив приведенные выше действия. Контейнер Tomcat должен функционировать до того, как вы захотите выполнить сервлет.

Запуск веб-браузера и запрос сервлета

Запустите веб-браузер и введите следующий адрес URL.

```
http://localhost:8080/examples/servlets/servlet/HelloServlet
```

В качестве альтернативы можно ввести такой адрес URL.

```
http://127.0.0.1:8080/examples/servlets/servlet/HelloServlet
```

Этот адрес URL допустим, поскольку IP-адрес 127.0.01 определен как адрес локального компьютера.

В окне браузера вы сможете увидеть выходные данные сервлета. В нем будет содержаться строка `Hello!` в полужирном начертании.

Интерфейс Servlet API

Описанные в этой главе классы и интерфейсы, необходимые для построения сервлетов, содержатся в двух пакетах — `javax.servlet` и `javax.servlet.http`. Они образуют интерфейс Servlet API. Имейте в виду, что эти пакеты не являются частью ключевых пакетов Java. Следовательно, они не включены в комплект Java SE. Наоборот, они предоставляются контейнером Tomcat, а также комплектом Java EE.

Интерфейс Servlet API находится в состоянии разработки и усовершенствования. Текущей спецификацией сервлета является версия 3.0, и именно она используется в примерах данной книги. Однако поскольку в мире Java все постоянно меняется, поинтересуйтесь, не появились ли какие-то дополнения или видоизменения. В этой главе обсуждается ядро интерфейса Servlet API, которое будет доступно большинству пользователей.

Пакет `javax.servlet`

Пакет `javax.servlet` содержит множество интерфейсов и классов, формирующих инфраструктуру, в рамках которой функционируют сервлеты. В табл. 32.1 представлены ключевые интерфейсы, предлагаемые в этом пакете. Самым главным из них является интерфейс `Servlet`. Все сервлеты должны реализовывать этот

интерфейс или расширять реализующий его класс. Интерфейсы `ServletRequest` и `ServletResponce` также являются очень важными.

Таблица 32.1. Ключевые интерфейсы пакета `javax.servlet`

Интерфейс	Описание
`Servlet`	Объявляет жизненный цикл методов для сервлета
`ServletConfig`	Позволяет сервлетам получать параметры инициализации
`ServletContext`	Позволяет сервлетам регистрировать события и обращаться к информации об их среде
`ServletRequest`	Используется для чтения данных из запроса клиента
`ServletResponse`	Используется для записи данных в ответ клиенту

В табл. 32.2 перечислены ключевые классы пакета `javax.servlet`.

Таблица 32.2. Ключевые классы пакета `javax.servlet`

Класс	Описание
`GenericServlet`	Реализует интерфейсы `Servlet` и `ServletConfig`
`ServletInputStream`	Предоставляет поток ввода для чтения запросов клиента
`ServletOutputStream`	Предоставляет поток вывода для записи ответов клиенту
`ServletException`	Указывает на то, что произошла ошибка сервлета
`UnavailableException`	Указывает на то, что сервлет является недоступным

Сейчас поговорим об этих интерфейсах и классах более подробно.

Интерфейс `Servlet`

Все сервлеты должны реализовать интерфейс `Servlet`. В нем объявлены методы `init()`, `service()` и `destroy()`, которые вызываются сервером во время жизненного цикла. Кроме них, предлагается также метод, позволяющий сервлету получать любые параметры инициализации. Методы, определяемые интерфейсом `Servlet`, перечислены в табл. 32.3.

Таблица 32.3. Методы, определенные интерфейсом `Servlet`

Метод	Описание
`void destroy()`	Вызывается при выгрузке сервлета
`ServletConfig getServletConfig()`	Возвращает объект интерфейса `ServletConfig`, содержащий любые параметры инициализации
`String getServletInfo()`	Возвращает строку, описывающую сервлет
`void init (ServletConfig sc) throws ServletException`	Вызывается во время инициализации сервлета. Параметры инициализации для сервлета могут быть получены из параметра *sc*. Если сервлет невозможно инициализировать, передается исключение `ServletException`
`void service(ServletRequest запр, ServletResponse рез) throws ServletException, IOException`	Вызывается для обработки запроса клиента. Запрос клиента можно прочитать из *запр*. Ответ клиенту можно записать в *рез*. В случае возникновения ошибок сервлета или ошибок ввода-вывода передается исключение

Методы `init()`, `service()` и `destroy()` определяют жизненный цикл сервлета. Они вызываются сервером. Метод `getServletConfig()` вызывается сервлетом для получения параметров инициализации. Разработчик сервлета переопределяет метод `getServletInfo()`, чтобы предоставить строку с полезной информацией (например, фамилия автора, номер версии, дата выпуска, авторские права). Этот метод также вызывается сервером.

Интерфейс `ServletConfig`

Интерфейс `ServletConfig` позволяет сервлету получать данные о конфигурации во время его загрузки. В табл. 32.4 перечислены методы, объявляемые этим интерфейсом.

Таблица 32.4. Методы, определенные интерфейсом `ServletConfig`

Метод	Описание
`ServletContext getServletContext()`	Возвращает содержимое для данного сервлета
`String getInitParameter(String` *параметр*`)`	Возвращает значение параметра инициализации (*параметр*)
`Enumeration getInitParameterNames()`	Возвращает перечень имен параметров инициализации
`String getServletName()`	Возвращает имя вызывающего сервлета

Интерфейс `ServletContext`

Интерфейс `ServletContext` позволяет сервлетам получать информацию об их среде. Некоторые его методы представлены в табл. 32.5.

Таблица 32.5. Методы, определенные интерфейсом `ServletContext`

Метод	Описание
`Object getAttribute(String` *атрибут*`)`	Возвращает значение атрибута сервера, указанного в параметре *атрибут*
`String getMimeType(String` *файл*`)`	Возвращает тип MIME файла
`String getRealPath(String` *виртПуть*`)`	Возвращает реальный путь, соответствующий виртуальному пути
`String getServerInfo()`	Возвращает информацию о сервере
`void log(String` *s*`)`	Записывает строку, указанную в соответствующем параметре, в журнал сервлета
`void log(String` *s*`, Throwable` *e*`)`	Записывает строку и трассировку стека для исключения в журнал сервлета
`void setAttribute(String` *атрибут*`, Object` *значение*`)`	Присваивает заданному атрибуту указанное значение

Интерфейс ServletRequest

Интерфейс ServletRequest позволяет сервлетам получать информацию о запросе клиента. Некоторые его методы представлены в табл. 32.6.

Таблица 32.6. Методы, определенные интерфейсом ServletRequest

Метод	Описание
Object getAttribute(String *атрибут*)	Возвращает значение указанного атрибута
String getCharacterEncoding()	Возвращает схему кодировки символов в запросе
Int getContentLength()	Возвращает размер запроса. Если размер невозможно определить, возвращается значение -1
String getContentType()	Возвращает тип запроса. Если тип невозможно определить, возвращается значение null
ServletInputStream getInputStream() throws IOException	Возвращает объект класса ServletInputStream, который можно использовать для чтения двоичных данных в запросе. Если метод getReader() уже был вызван для этого запроса, передается исключение IllegalStateException
String getParameter(String *имяПарам*)	Возвращает значение указанного параметра
Enumeration<String> getParameterNames()	Возвращает перечень имен параметров для данного запроса
String[] getParameterValues(String *имя*)	Возвращает массив, состоящий из значений, связанных с параметром *имя*
String getProtocol()	Возвращает описание протокола
BufferedReader getReader() throws IOException	Возвращает буферизированный читатель, который можно использовать для чтения текста из запроса. Если метод getInputStream() уже был вызван для данного запроса, передается исключение IllegalStateException
String getRemoteAddr()	Возвращает строковый эквивалент IP-адреса клиента
String getRemoteHost()	Возвращает строковый эквивалент имени хоста клиента
String getScheme()	Возвращает схему передачи адреса URL, которая используется для запроса (например, "http", "ftp")
String getServerName()	Возвращает имя сервера
int getServerPort()	Возвращает номер порта

Интерфейс ServletResponce

Интерфейс ServletResponce позволяет сервлету формулировать ответ клиенту. Некоторые его методы описаны в табл. 32.7.

Таблица 32.7. Методы, определенные интерфейсом ServletResponce

Метод	Описание
`String getCharacterEncoding()`	Возвращает схему кодировки символов в ответе
`ServletOutputStream getOutputStream() throws IOException`	Возвращает объект класса `ServletOutputStream`, который можно использовать для записи двоичных данных в ответ. Если метод `gerWriter()` уже был вызван для данного запроса, передается исключение `IllegalStateException`
`PrintWriter getWriter() throws IOException`	Возвращает объект класса `PrintWriter`, который можно использовать для записи символьных данных в ответ. Если метод `gerOutputStream()` уже был вызван для данного запроса, передается исключение `IllegalStateException`
`void setContentLength(int` *размер*`)`	Задает размер содержимого для ответа
`void setContentType(String` *тип*`)`	Задает тип содержимого для ответа

Класс GenericServlet

Класс `GenericServlet` предлагает реализации основных методов жизненного цикла сервлета. Этот класс реализует интерфейсы `Servlet` и `ServletConfig`. Кроме того, доступен также и метод для добавления строки в журнал сервера. Ниже показаны сигнатуры этого метода.

```
void log(String s)
void log(String s, Throwable e)
```

Здесь *s* — это строка, которую необходимо добавить в журнал, а *e* — это передаваемое исключение.

Класс ServletInputStream

Класс `ServletInputStream` расширяет класс `InputStream`. Он реализуется контейнером сервлета и предлагает входной поток, который разработчик сервлета может использовать для чтения данных из запроса клиента. Он определяет конструктор, применяемый по умолчанию. Кроме того, предлагается метод для чтения байтов из потока. Ниже показана его сигнатура.

```
int readLine(byte[] буфер, int смещение, int размер) throws IOException
```

Здесь *буфер* — это массив, в котором хранится определенное количество (*размер*) байтов, начиная со смещения (*смещение*). Метод возвращает фактическое количество прочитанных байтов или значение `-1`, если возникнет условие окончания потока.

Класс ServletOutputStream

Класс `ServletOutputStream` расширяет класс `OutputStream`. Он реализуется контейнером сервлета и предлагает выходной поток, который разработчик сервлета может применять для записи данных в ответ клиенту. Определяется конструктор,

используемый по умолчанию. Он также определяет методы `print()` и `println()`, которые выводят данные в поток.

Класс `ServletException`

Пакет `javax.servlet` определяет два исключения. Первый из них — исключение `ServletException`, которое свидетельствует об ошибке сервлета. Вторым является `UnavailableException`, класс которого расширяет класс `ServletException`. Это исключение свидетельствует о том, что сервлет недоступен.

Чтение параметров сервлета

Интерфейс `ServletRequest` включает методы, позволяющие считывать имена и значения параметров, включенных в запрос клиента. Мы займемся созданием сервлета, который продемонстрирует их использование. Пример содержит два файла. Веб-страница определена в файле `PostParameters.html`, а сервлет — в файле `PostParametersServlet.java`.

В следующем листинге приведен исходный код HTML файла `PostParameters.html`. Этот код определяет таблицу, состоящую из двух меток и двух текстовых полей. Одной из меток является Employee (Сотрудник), а другой — Phone (Телефон). Имеется также кнопка подтверждения. Обратите внимание на то, что параметр `action` дескриптора формы (`<form>`) определяет адрес URL. Этот адрес идентифицирует сервлет, который будет выполнять обработку запроса HTTP `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>
```

В следующем листинге представлен исходный код файла `PostParameters Servlet.java`. Метод `service()` переопределяется для обработки запросов клиентов. Метод `getParameterNames()` возвращает перечень имен параметров. Их обработка осуществляется в цикле. Как видите, клиенту выводится имя параметра и его значение. Получение значения параметра производится с помощью метода `getParameter()`.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
```

```
public class PostParametersServlet
extends GenericServlet {

    public void service(ServletRequest request,
                        ServletResponse response)
    throws ServletException, IOException {

        // Получаем класс PrintWriter.
        PrintWriter pw = response.getWriter();

        // Получаем перечень имен параметров.
        Enumeration e = request.getParameterNames();

        // Отображаем имена параметров и их значения.
        while(e.hasMoreElements()) {
            String pname = (String)e.nextElement();
            pw.print(pname + " = ");
            String pvalue = request.getParameter(pname);
            pw.println(pvalue);
        }
        pw.close();
    }
}
```

Скомпилируйте сервлет. Затем скопируйте его в соответствующий каталог и обновите файл `web.xml`, как упоминалось ранее. После этого выполните следующие действия для проверки примера.

1. Запустите контейнер Tomcat (если это еще не сделано).
2. Отобразите веб-страницу в окне браузера.
3. Введите в текстовых полях фамилию служащего и номер телефона.
4. Отправьте веб-страницу.

После этого в окне браузера будет отображен ответ, динамически созданный сервлетом.

Пакет `javax.servlet.http`

Для иллюстрации основных функциональных возможностей сервлетов в приведенных примерах использовались такие классы и интерфейсы из пакета `javax.servlet`, как `ServletRequest`, `ServletResponse` и `GenericServlet`. Но при работе с протоколом HTTP вы будете обычно использовать интерфейсы и классы из пакета `javax.servlet.http`. Как вы увидите, его функциональные возможности облегчают создание сервлетов, которые работают с запросами и ответами HTTP.

В табл. 32.8 перечислены основные интерфейсы, которые предлагает этот пакет.

Таблица 32.8. Основные интерфейсы пакета `javax.servlet.http`

Интерфейс	Описание
`HttpServletRequest`	Позволяет сервлетам считывать данные из запроса HTTP
`HttpServletResponse`	Позволяет сервлетам записывать данные в ответ HTTP
`HttpSession`	Позволяет считывать и записывать данные сеансов
`HttpSessionBindingListener`	Информирует объект о том, связан ли он с сеансом

В табл. 32.9 описаны основные классы, предлагаемые в этом пакете. Наиболее важным из них является класс `HttpServlet`. Разработчики сервлетов обычно расширяют этот класс для обработки запросов HTTP.

Таблица 32.9. Основные классы пакета `javax.servlet.http`

Класс	Описание
`Cookie`	Позволяет хранить информацию о состоянии на компьютере клиента
`HttpServlet`	Предлагает методы для обработки запросов и ответов HTTP
`HttpSessionEvent`	Инкапсулирует событие изменения сеанса
`HttpSessionBindingEvent`	Показывает, связан ли слушатель со значением сеанса, или демонстрирует, что атрибут сеанса был изменен

Интерфейс `HttpServletRequest`

Интерфейс `HttpServletRequest` реализуется контейнером сервлета. Он позволяет сервлету получать информацию о запросе клиента. В табл. 32.10 перечислены некоторые его методы.

Таблица 32.10. Методы, определенные в интерфейсе `HttpServletRequest`

Метод	Описание
`String getAuthType()`	Возвращает схему аутентификации
`Cookie[] getCookies()`	Возвращает массив, содержащий файл cookie в данном запросе
`long getDateHeader(String` *поле*`)`	Возвращает значение поля заголовка даты
`String getHeader(String` *поле*`)`	Возвращает значение поля заголовка
`Enumeration<String> getHeaderNames()`	Возвращает перечень имен заголовков
`int getIntHeader(String` *поле*`)`	Возвращает целочисленный (`int`) эквивалент поля заголовка
`String getMethod()`	Возвращает метод HTTP для запроса
`String getPathInfo()`	Возвращает любую информацию о маршруте, который определен после маршрута сервлета и перед строкой запроса в адресе URL
`String getPathTranslated()`	Возвращает любую информацию о маршруте, который определен после маршрута сервлета и перед строкой запроса в адресе URL, после перевода его в действительный маршрут
`String getQueryString()`	Возвращает любой строковый запрос в адрес URL
`String getRemoteUser()`	Возвращает имя пользователя, который создал данный запрос
`String getRequestedSessionId()`	Возвращает идентификатор сеанса
`String getRequestURI()`	Возвращает адрес URL

Окончание табл. 32.10

Метод	Описание
`StringBuffer getRequestURL()`	Возвращает адрес URL
`String getServletPath()`	Возвращает часть адреса URL, которая идентифицирует сервлет
`HttpSession getSession()`	Возвращает сеанс для данного запроса. Если сеанс не существует, он создается, а затем возвращается
`HttpSession getSession(Boolean` *новый*`)`	Если значением параметра *новый* является `true` и сеанса не существует, метод создает и возвращает сеанс для данного запроса. В противном случае он возвращает существующий сеанс для данного запроса
`boolean isRequested SessionIdFromCookie()`	Возвращает значение `true`, если файл cookie содержит идентификатор сеанса. В противном случае возвращает значение `false`
`boolean isRequested SessionIdFromURL()`	Возвращает значение `true`, если адрес URL содержит идентификатор сеанса. В противном случае возвращает значение `false`
`boolean isRequested SessionIdValid()`	Возвращает значение `true`, если запрошенный идентификатор сеанса является действительным в текущем содержимом сеанса

Интерфейс `HttpServletResponse`

Интерфейс `HttpServletResponse` позволяет сервлету сформулировать для клиента ответ HTTP. Он определяет несколько констант. Они соответствуют кодам различных состояний, которые можно назначать ответу HTTP. Например, значение `SC_OK` показывает, что ответ HTTP достиг цели, а значение `SC_NOT_FOUND` — что запрошенный ресурс является недоступным. В табл. 32.11 приводятся некоторые методы этого интерфейса.

Таблица 32.11. Методы, определенные в интерфейсе `HttpServletResponse`

Метод	Описание
`void addCookie(Cookie` *cookie*`)`	Добавляет *cookie* в ответ HTTP
`boolean containsHeader(String` *поле*`)`	Возвращает значение `true`, если в заголовке ответа HTTP содержится заданное поле
`String encodeURL(String` *url*`)`	Определяет, должен ли идентификатор сеанса быть закодированным в адресе URL. Если да, возвращает измененную версию адреса URL. В противном случае возвращает адрес URL. Все адреса URL, созданные сервером, должны обрабатываться этим методом
`String encodeRedirectURL(String` *url*`)`	Определяет, должен ли идентификатор сеанса быть закодированным в адресе URL. Если да, возвращает измененную версию адреса URL. В противном случае возвращает адрес URL. Все адреса URL, переданные методу `sendRedirect()`, должны обрабатываться этим методом
`void sendError(int` *c*`) throws IOException`	Посылает клиенту заданный код ошибки

Окончание табл. 32.11

Метод	Описание
`void sendError(int c, String s) throws IOException`	Посылает клиенту заданный код ошибки и строку сообщения
`void sendRedirect(String url) throws IOException`	Переадресовывает клиента по указанному адресу URL
`void setDateHeader(String поле, long миллисекунд)`	Добавляет *поле* в заголовок со значением даты, исчисляемой в миллисекундах (*миллисекунд*). Отсчет миллисекунд ведется с полуночи 1 января 1970 года (GMT)
`void setHeader(String поле, String значение)`	Добавляет *поле* в заголовок со значением, указанным в параметре *значение*
`void setIntHeader(String поле, int значение)`	Добавляет *поле* в заголовок со значением, указанным в параметре *значение*
`void setStatus(int код)`	Присваивает заданный код состоянию для данного ответа

Интерфейс `HttpSession`

Интерфейс `HttpSession` позволяет сервлету считывать и записывать информацию о состоянии, связанную с сеансом HTTP. Некоторые из его методов перечислены в табл. 32.12. Каждый из них передает исключение `IllegalStateException`, если сеанс уже является недействительным.

Таблица 32.12. Методы, определенные в интерфейсе `HttpSession`

Метод	Описание
`Object getAttribute(String атрибут)`	Возвращает значение, связанное с именем (параметр *атрибут*). Возвращает значение `null`, если указанный атрибут не был найден
`Enumeration<String> getAttributeNames()`	Возвращает перечень имен атрибутов, связанных с сеансом
`long getCreationTime()`	Возвращает время, прошедшее с момента создания сеанса. Отсчет ведется в миллисекундах, начиная с полуночи 1 января 1970 года (GMT)
`String getId()`	Возвращает идентификатор сеанса
`long getLastAccessedTime()`	Возвращает время, прошедшее с того момента, когда клиент произвел последний запрос для данного сеанса. Отсчет времени ведется в миллисекундах, начиная с полуночи 1 января 1970 года (GMT)
`void invalidate()`	Отменяет данный сеанс и удаляет его из содержимого
`boolean isNew()`	Возвращает значение `true`, если сервер создал сеанс, к которому еще не обращался клиент
`void removeAttribute(String атрибут)`	Удаляет из сеанса заданный атрибут
`void setAttribute(String атрибут, Object значение)`	Связывает заданное значение с именем заданного атрибута

Интерфейс HttpSessionBindingListener

Интерфейс HttpSessionBindingListener реализуется объектами, для которых необходимо уведомление о том, являются ли они связанными или несвязанными с сеансом HTTP. Ниже перечислены методы, которые вызываются, если объект связан или, наоборот, не связан.

```
void valueBound(HttpSessionBindingEvent e)
void valueUnbound(HttpSessionBindingEvent e)
```

Здесь *e* — объект события, описывающий связывание.

Класс Cookie

Класс Cookie инкапсулирует файл cookie. Файл cookie — это строки с данными, которые хранятся на компьютере клиента и содержат информацию о состоянии. Файлы cookie полезны для отслеживания активности пользователей. Например, предположим, что пользователь посещает интерактивный магазин. Файл cookie может хранить имя пользователя, адрес и другую информацию. Пользователю не нужно будет вводить эти данные каждый раз при посещении магазина.

Сервлет может сохранить файл cookie на компьютере пользователя с помощью метода addCookie() интерфейса HttpServletResponce. Данные из этого файла затем включаются в заголовок ответа HTTP, который отправляется браузеру.

Имена и значения файла cookie хранятся на компьютере пользователя. Часть информации, сохраняемой для каждого файла cookie, включает следующие сведения:

- имя файла cookie;
- значение файла cookie;
- истечение срока действия файла cookie;
- домен и путь к файлу cookie.

Истечение срока действия определяет, когда данный файл cookie будет удален из компьютера пользователя. Если эта дата не назначена явным образом, файл cookie удаляется по завершении текущего сеанса браузера. В противном случае он сохраняется в файле на компьютере пользователя.

Домен и путь к файлу cookie определяют, когда он будет включен в заголовок запроса HTTP. Если пользователь вводит адрес URL, домен и путь которого совпадают с этими значениями, файл cookie назначается веб-серверу, в противном случае — нет.

Для класса Cookie предусмотрен один конструктор. Он имеет следующую сигнатуру.

```
Cookie(String имя, String значение)
```

Здесь *имя* и *значение* файла cookie передаются конструктору в качестве параметров. Методы класса Cookie перечислены в табл. 32.13.

Таблица 32.13. Методы, определяемые классом Cookie

Метод	Описание
Object clone()	Возвращает копию этого объекта
String getComment()	Возвращает комментарий
String getDomain()	Возвращает домен
int getMaxAge()	Возвращает максимальный возраст (в секундах)

Окончание табл. 32.13

Метод	Описание
`String getName()`	Возвращает имя
`String getPath()`	Возвращает маршрут
`boolean getSecure()`	Возвращает значение `true` для безопасного файла cookie. В противном случае возвращает значение `false`
`String getValue()`	Возвращает значение
`int getVersion()`	Возвращает версию
`boolean isHttpOnly()`	Возвращает значение `true`, если файл cookie содержит атрибут `HttpOnly`
`void setComment(String c)`	Присваивает заданный комментарий
`void setDomain(String d)`	Присваивает заданный домен
`void setComment(String` `void setHttpOnly(boolean` *`толькоHttp`*`)`	Если параметр *`толькоHttp`* содержит значение `true`, то атрибут `HttpOnly` добавляется к файлу cookie. Если параметр *`толькоHttp`* содержит значение `false`, атрибут `HttpOnly` удаляется
`void setMaxAge(int` *`секунд`*`)`	Устанавливает максимальный возраст файла cookie в секундах. Это значение представляет собой количество секунд, по истечении которых файл cookie будет удален
`void setPath(String p)`	Присваивает заданный путь
`void setSecure(Boolean` *`защищен`*`)`	Присваивает заданный флаг безопасности
`void setValue(String v)`	Присваивает заданное значение
`void setVersion(int v)`	Присваивает заданную версию

Класс `HttpServlet`

Класс `HttpServlet` расширяет класс `GenericServlet`. Обычно он используется при разработке сервлетов, получающих и обрабатывающих запросы HTTP. Методы класса `HttpServlet` представлены в табл. 32.14.

Таблица 32.14. Методы, определяемые классом `HttpServlet`

Метод	Описание
`void doDelete(HttpServletRequest` *`запр`*`, HttpServletResponse` *`рез`*`) throws IOException, ServletException`	Обрабатывает запрос HTTP `DELETE`
`void doGet(HttpServletRequest` *`запр`*`, HttpServletResponse` *`рез`*`) throws IOException, ServletException`	Обрабатывает запрос HTTP `GET`
`void doOptions(HttpServletRequest` *`запр`*`, HttpServletResponse` *`рез`*`) throws IOException, ServletException`	Обрабатывает запрос HTTP `OPTIONS`

Окончание табл. 32.14

Метод	Описание
`void doPost(HttpServletRequest` *`запр`*`, HttpServletResponse` *`рез`*`) throws IOException, ServletException`	Обрабатывает запрос HTTP `POST`
`void doPut(HttpServletRequest` *`запр`*`, HttpServletResponse` *`рез`*`) throws IOException, ServletException`	Обрабатывает запрос HTTP `PUT`
`void doTrace(HttpServletRequest` *`запр`*`, HttpServletResponse` *`рез`*`) throws IOException, ServletException`	Обрабатывает запрос HTTP `TRACE`
`Long getLastModified (HttpServletRequest` *`запр`*`)`	Возвращает время, измеряемое в миллисекундах после полуночи 1 января 1970 года (GMT), когда в последний раз был изменен запрошенный ресурс
`void service(HttpServletRequest` *`запр`*`, HttpServletResponse` *`рез`*`) throws IOException, ServletException`	Вызывается сервером, когда для данного сервлета поступает запрос HTTP. Параметры предусматривают доступ к запросу и ответу HTTP соответственно

Класс `HttpSessionEvent`

Класс `HttpSessionEvent` инкапсулирует события сеансов. Он расширяет класс `EventObject` и создается во время изменений, производимых в сеансе. В классе `HttpSessionEvent` определен следующий конструктор.

```
HttpSessionEvent(HttpSession сеанс)
```

Здесь *`сеанс`* — это источник извещения о событии.

Класс `HttpSessionEvent` определяет один метод, `getSession()`, который показан ниже.

```
HttpSession getSession()
```

Он возвращает сеанс, в котором произошло событие.

Класс `HttpSessionBindingEvent`

Класс `HttpSessionBindingEvent` расширяет класс `HttpSessionEvent`. Он создается в том случае, когда слушатель связан или не связан со значением в объекте интерфейса `HttpSession`. Он создается также, если атрибут является связанным или несвязанным. Ниже показаны его конструкторы.

```
HttpSessionBindingEvent(HttpSession сеанс, String имя)
HttpSessionBindingEvent(HttpSession сеанс, String имя, Object значение)
```

Здесь *`сеанс`* — это источник извещения о событии, а *`имя`* — имя связанного или несвязанного объекта. Если параметр связан или не связан, ему передается заданное значение.

Метод `getName()` получает связанное или несвязанное имя. Ниже показан его конструктор.

```
String getName()
```

Показанный ниже метод `getSession()` получает сеанс, с которым связан или не связан слушатель.

```
HttpSession getSession()
```

Метод `getValue()` получает значение связанного или несвязанного параметра.

```
Object getValue()
```

Обработка запросов и ответов HTTP

Класс `HttpServlet` предлагает специализированные методы, обрабатывающие различные типы запросов HTTP. Разработчики сервлетов обычно переопределяют один из этих методов. К этим методам относятся `doDelete()`, `doGet()`, `doHead()`, `doOptions()`, `doPost()`, `doPut()` и `doTrace()`. Привести полное описание различных типов запросов HTTP в этой книге невозможно. Однако чаще всего при обработке форм используются запросы `GET` и `POST`, потому в этом разделе приводятся примеры этих случаев.

Обработка запросов HTTP `GET`

Сейчас займемся разработкой сервлета, обрабатывающего запрос HTTP `GET`. Сервлет вызывается при подтверждении заполнения формы на веб-странице. В примере задействовано два файла. Веб-страница определена в файле `ColorGet.html`, а сервлет — в файле `ColorGetServlet.java`. Исходный код HTML файла `ColorGet.html` показан в следующем листинге. Он определяет форму, содержащую элемент выбора и кнопку для оправки. Обратите внимание на то, что параметр `action` дескриптора `<form>` определяет адрес URL. Этот адрес идентифицирует сервлет для обработки запроса HTTP `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>
```

В следующем листинге показан исходный код файла `ColorGetServlet.java`. Метод `doGet()` переопределяется для обработки любых запросов HTTP `GET`, посылаемых данному сервлету. Для получения выбора, сделанного пользователем, он использует метод `getParameter()` интерфейса `HttpServletRequest`. После этого формируется ответ.

```
import java.io.*;
import javax.servlet.*;
import javax.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: ");
        pw.println(color);
        pw.close();
    }
}
```

Скомпилируйте сервлет. Затем скопируйте его в соответствующий каталог и обновите файл `web.xml`, о чем было сказано ранее. Далее выполните следующие действия для проверки этого примера.

1. Запустите контейнер Tomcat, если он еще не работает.
2. Отобразите веб-страницу в окне браузера.
3. Выберите цвет.
4. Отправьте форму на веб-странице.
5. После этого браузер отобразит ответ, динамически созданный сервлетом.

Еще один момент: параметры для запроса HTTP `GET` включены как часть адреса URL, посылаемого веб-серверу. Предположим, что пользователь выбрал красный цвет и отправил форму. Адрес URL, который посылается серверу из браузера, будет иметь следующий вид.

```
http://localhost:8080/examples/servlets/servlet/
ColorGetServlet?color=Red
```

Символы, стоящие справа от вопросительного знака, называются *строкой запроса.*

Обработка запросов HTTP `POST`

Теперь перейдем к разработке сервлета, обрабатывающего запрос HTTP `POST`. Сервлет вызывается при подтверждении формы на веб-странице. В примере задействовано два файла. Веб-страница определена в файле `ColorPost.html`, а сервлет — в файле `ColorPostServlet.java`.

В следующем листинге показан исходный код HTML в файле `ColorPost.html`. Он идентичен коду файла `ColorGet.html`, за исключением того, что параметр `method` дескриптора `<form>` явным образом определяет, что следует использовать метод `POST`, а в параметре `action` того же дескриптора указан 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>
```

В следующем листинге показан исходный код файла `ColorPostServlet.java`. Метод `doPost()` заменяется для обработки любых запросов HTTP `POST`, отправляемых данному сервлету. Для получения сделанного пользователем выбора он использует метод `getParameter()` интерфейса `HttpServletRequest`. После этого формируется ответ.

```
import java.io.*;
import javax.servlet.*;
import javax.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: ");
        pw.println(color);
        pw.close();
    }
}
```

Скомпилируйте сервлет. Чтобы проверить его, выполните те же действия, что и в предыдущем разделе.

На заметку! Параметры для запроса HTTP `POST` не включаются в адрес URL, отправляемый веб-серверу. В этом примере браузер посылает серверу следующий адрес URL: `http://localhost:8080/examples/servlets/servlet/ColorPostServlet`. Имена параметров и значения отправляются в теле запроса HTTP.

Использование файлов cookie

Теперь давайте разработаем сервлет, который поможет проиллюстрировать процесс использования файлов cookie. Сервлет вызывается при отправке формы на веб-странице. Пример содержит три файла, описанных в табл. 32.15.

Таблица 32.15. Примеры применения файлов cookie

Файл	Описание
`AddCookie.html`	Позволяет пользователю определять значение для файла cookie по имени `MyCookie`
`AddCookieServlet.java`	Обрабатывает отправку файла `AddCookie.html`
`GetCookiesServlet.java`	Отображает значения файла cookie

В следующем листинге показан исходный код HTML файла AddCookie.html. Эта страница содержит текстовое поле для ввода значения. На странице имеется кнопка отправки. Если щелкнуть на этой кнопке, значение в текстовом поле будет отправлено AddCookieServlet при помощи запроса HTTP 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>
```

В следующем листинге показан исходный код файла AddCookieServlet.java. Он получает значение параметра по имени data. Затем он создает объект MyCookie класса Cookie, который содержит значение параметра data. После этого в заголовок ответа HTTP добавляется файл cookie с помощью метода addCookie(). Далее браузер получает ответное сообщение.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AddCookieServlet extends HttpServlet {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
    throws ServletException, IOException {

        // Получение параметра от запроса HTTP.
        String data = request.getParameter("data");

        // Создание файла cookie.
        Cookie cookie = new Cookie("MyCookie", data);

        // Добавление файла cookie в ответ HTTP.
        response.addCookie(cookie);

        // Запись вывода в браузер.
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>MyCookie has been set to");
        pw.println(data);
        pw.close();
    }
}
```

В следующем листинге показан исходный код файла GetCookieServlet.java. Он вызывает метод getCookie() для чтения любых файлов cookie, включенных в запрос HTTP GET. Имена и значения этих файлов cookie включаются в ответ HTTP. Обратите внимание на то, что для получения этой информации вызываются методы getName() и getValue().

```
import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;

public class GetCookiesServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
    throws ServletException, IOException {

        // Получение файлов cookie из заголовка запроса HTTP.
        Cookie[] cookies = request.getCookies();

        // Отображение всех файлов cookie.
        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 эти сервлеты. Затем скопируйте их в соответствующий каталог и обновите файл `web.xml`, как упоминалось ранее. После этого выполните следующие действия, чтобы проверить данный пример.

1. Запустите контейнер Tomcat, если он еще не работает.
2. Отобразите файл `AddCookie.html` в окне браузера.
3. Введите значение для объекта `MyCookie`.
4. Отправьте форму на веб-странице.

После выполнения этих примеров вы увидите, что в окне браузера отображено ответное сообщение.

В строке адреса браузера введите следующий адрес URL.

`http://localhost:8080/examples/servlets/servlet/GetCookiesServlet`

Обратите внимание на то, что в окне браузера отображаются имя и значение файлов cookie.

В этом примере не применяется метод `setMaxAge()` класса `Cookie` для явного назначения истечения срока действия файлов cookie. Поэтому срок действия файлов cookie истекает по завершении сеанса браузера. Если использовать метод `setMaxAge()`, то вы увидите, что файл cookie будет сохранен на диске клиентского компьютера.

Отслеживание сеансов

Протокол HTTP не поддерживает состояние. Каждый запрос не зависит от предыдущего запроса. Однако в некоторых приложениях иногда необходимо сохранять информацию о состоянии, которая потом будет анализироваться после общения браузера и сервера. Такой механизм предлагают сеансы.

Сеанс можно создать с помощью метода `getSession()` интерфейса `HttpServletRequest`. Возвращается объект интерфейса `HttpSession`. Этот

объект способен сохранять набор связей между именами и объектами. Этими связями управляют методы `setAttribute()`, `getAttribute()`, `getAttributeNames()` и `removeAttribute()` интерфейса `HttpSession`. Состояние сеанса совместно используется всеми сервлетами, связанными с определенным клиентом.

Следующий сервлет иллюстрирует использование информации о состоянии сеанса. Метод `getSession()` получает текущий сеанс. Если сеанс не существует, создается новый сеанс. Метод `getAttribute()` вызывается для получения объекта, связанного с именем `date`. Этим объектом является объект класса `Date`, инкапсулирующий дату и время последнего доступа к данной странице. (Естественно, такой связи не будет, если доступ к странице производится в первый раз.) Затем создается объект класса `Date`, инкапсулирующий текущую дату и время. Метод `setAttribute()` вызывается для связывания имени `date` с этим объектом.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DateServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
    throws ServletException, IOException {

        // Получение объекта HttpSession.
        HttpSession hs = request.getSession(true);

        // Получение класса PrintWriter.
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.print("<B>");

        // Отображение даты/времени последнего доступа.
        Date date = (Date)hs.getAttribute("date");
        if(date != null) {
            pw.print("Last access: " + date + "<br>");
        }

        // Отображение текущей даты/времени.
        date = new Date();
        hs.setAttribute("date", date);
        pw.println("Current date: " + date);
    }
}
```

Когда впервые запросите этот сервлет, браузер отобразит одну строку с информацией о текущих дате и времени. При последующем вызове будут отображаться две строки. В первой строке будет указана дата и время последнего доступа к сервлету, а во второй — текущие дата и время.

ЧАСТЬ IV

Применение Java

ГЛАВА

Финансовые аплеты и сервлеты

Помимо больших и сложных приложений, к числу которых относятся текстовые процессоры, базы данных и пакеты программ бухгалтерского учета и которые доминируют в мире вычислений, существует класс программ, которые являются одновременно и популярными, и небольшими. Они предназначены для выполнения различных финансовых расчетов – регулярных платежей по ссуде, будущей стоимости вклада, остатка баланса по ссуде. Ни один из этих расчетов не является сложным и не требует множества строк кода, а получаемая с их помощью информация является очень полезной.

Как вы знаете, язык Java изначально предназначался для создания небольших переносимых программ. Сначала эти программы принимали форму аплетов, однако спустя несколько лет появились сервлеты. (Напомним, что *аплеты* выполняются на локальном компьютере, внутри браузера, а *сервлеты* функционируют на сервере.) Большинство обычных финансовых расчетов, вследствие их небольших размеров, удобно производить в сервлетах и аплетах. Более того, если финансовый аплет/сервлет добавить на веб-страницу, то пользователи наверняка сочтут это удобным. Они будут регулярно посещать страницу, на которой можно произвести необходимый расчет.

В этой главе займемся разработкой некоторых аплетов, производящих следующие финансовые расчеты:

- регулярные платежи по ссуде;
- остаток баланса по ссуде;
- будущая стоимость вклада;
- первоначальная сумма вклада, необходимая для достижения желаемой будущей стоимости;
- годовой доход по вкладу;
- сумма вклада, необходимая для достижения желаемого годового дохода.

В конце главы будет показан способ преобразования финансовых аплетов в сервлеты.

Расчет платежей по ссуде

Пожалуй, самым распространенным финансовым расчетом является расчет регулярных платежей по ссуде, выданной, например, на покупку автомобиля или жилого дома. Расчет платежей по ссуде производится с помощью следующей формулы.

```
Payment = (intRate * (principal / payPerYear)) /
          (1 - ((intRate / payPerYear) + 1)-payPerYear*numYears)
```

Здесь *intRate* — процент по ссуде, *principal* — первоначальный баланс, *payPerYear* — количество платежей в течение одного года, а *numYear* — срок погашения ссуды в годах.

Следующий аплет, RegPay, использует приведенную выше формулу для расчета платежей по ссуде с учетом информации, введенной пользователем. Как и остальные аплеты в этой главе, аплет RegPay основан на библиотеке Swing. Это означает, что он расширяет класс JApplet и использует классы библиотеки Swing для создания пользовательского интерфейса. Обратите внимание также на то, что этот класс реализует интерфейс ActionListener.

```
// Простой аплет для расчетов по ссуде.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
<applet code="RegPay" width=320 height=200>
</applet>
*/

public class RegPay extends JApplet
implements ActionListener {

    JTextField amountText, paymentText, periodText, rateText;
    JButton doIt;

    double principal; // первоначальная сумма
    double intRate;   // процент по ссуде
    double numYears;  // срок погашения ссуды в годах

    /* Количество платежей в течение одного года. Можно сделать
    так, чтобы это значение задавал пользователь. */
    final int payPerYear = 12;

    NumberFormat nf;

    public void init() {
        try {
            SwingUtilities.invokeAndWait(new Runnable () {
                public void run() {
                    makeGUI(); // инициализация GUI
                }
            });
        } catch(Exception exc) {
            System.out.println("Can't create because of "+ exc);
         // System.out.println("Невозможно создать из-за "+ exc);
        }
    }

    // Устанавливаем и инициализируем GUI.
    private void makeGUI() {
        // Используем сеточную компоновку.
        GridBagLayout gbag = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(gbag);
        JLabel heading = new
                    JLabel("Compute Monthly Loan Payments");
        // JLabel heading = new
```

```
// JLabel("Вычисление ежемесячных платежей по ссуде");

JLabel amountLab = new JLabel("Principal ");
// JLabel amountLab = new JLabel("Начальный баланс ");
JLabel periodLab = new JLabel("Years ");
// JLabel periodLab = new JLabel("Количество лет ");
JLabel rateLab = new JLabel("Interest Rate ");
// JLabel rateLab = new JLabel("Процент по ссуде ");
JLabel paymentLab = new JLabel("Monthly Payments ");
// JLabel paymentLab = new JLabel("Ежемесячный платеж ");

amountText = new JTextField(10);
periodText = new JTextField(10);
paymentText = new JTextField(10);
rateText = new JTextField(10);

// Поле платежа только для отображения.
paymentText.setEditable(false);
doIt = new JButton("Compute");
// doIt = new JButton("Вычислить");

// Определяем сетку.
gbc.weighty = 1.0; // используем строку, весом 1
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);
// Располагаем большинство компонентов справа.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(amountLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(amountText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(paymentLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(paymentText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

// Добавляем все компоненты.
add(heading);
add(amountLab);
add(amountText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(paymentLab);
```

```
        add(paymentText);
        add(doIt);

        // Регистрируем на получение событий действий.
        amountText.addActionListener(this);
        periodText.addActionListener(this);
        rateText.addActionListener(this);
        doIt.addActionListener(this);

        // Создаем формат числа.
        nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(2);
        nf.setMaximumFractionDigits(2);
    }
    /* Пользователь нажал <Enter> в текстовом поле или щелкнул
    на кнопке Compute. Отображаем результат, если все
    поля заполнены. */
    public void actionPerformed(ActionEvent ae) {
        double result = 0.0;

        String amountStr = amountText.getText();
        String periodStr = periodText.getText();
        String rateStr = rateText.getText();
        try {
            if(amountStr.length() != 0 &&
               periodStr.length() != 0 &&
                 rateStr.length() != 0) {

                principal = Double.parseDouble(amountStr);
                numYears = Double.parseDouble(periodStr);
                intRate = Double.parseDouble(rateStr) / 100;
                result = compute();
                paymentText.setText(nf.format(result));
            }
            showStatus(""); // удаляем любое предыдущее сообщение
                            // об ошибке
        } catch (NumberFormatException exc) {

            showStatus("Invalid Data");
         // showStatus("Неверные данные");
            paymentText.setText("");
        }
    }
    // Расчет платежа по ссуде.
    double compute() {
        double numer;
        double denom;
        double b, e;

        numer = intRate * principal / payPerYear;

        e = -(payPerYear * numYears);
        b = (intRate / payPerYear) + 1.0;

        denom = 1.0 - Math.pow(b, e);
        return numer / denom;
    }
}
```

Аплет, созданный этой программой, показан на рис. 33.1. Чтобы проверить его в действии, введите сумму, полученную по ссуде, срок погашения ссуды и процент

по ссуде. Предполагается, что платежи будут производиться ежемесячно. Как только информация будет введена, щелкните на кнопке Compute (Вычислить), чтобы рассчитать сумму ежемесячного платежа.

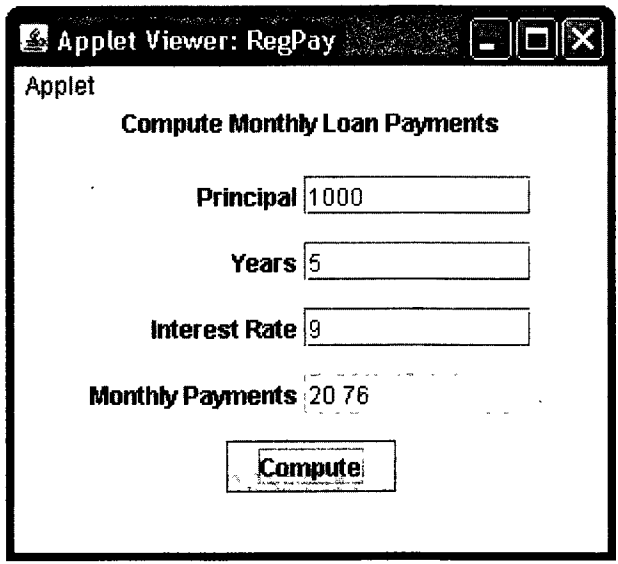


Рис. 33.1. Окно аплета `RegPay`

В следующих разделах подробно анализируется код аплета `RegPay`. Поскольку все аплеты в этой главе используют одну и ту же базовую структуру, большая часть изложенного здесь объяснения применима и к другим аплетам в этой главе.

Поля аплета `RegPay`

Класс аплета `RegPay` начинается с объявления переменных экземпляра, содержащих ссылки на текстовые поля, в которых пользователь будет вводить информацию о ссуде. Затем в нем объявляется переменная `doIt`, которая будет хранить ссылку на кнопку Compute.

Класс аплета `RegPay` объявляет три переменные типа `double`, которые хранят значения по ссуде. Основная сумма ссуды хранится в переменной `principal`, процент по ссуде — в переменной `intRate`, а срок погашения ссуды (измеряется в годах) — в переменной `numYears`. Эти значения пользователь вводит в текстовых полях. После этого объявляется последняя финальная целочисленная переменная `payPerYear`, которой присваивается значение 12. Таким образом, количество платежей в течение одного года получается жестко запрограммированным — 12 платежей, или ежемесячно в течение года, поскольку на практике большинство ссуд оформляется именно так. Как следует из комментариев, можно сделать так, чтобы пользователь самостоятельно вводил это значение, хотя для этого потребуется еще одно текстовое поле.

Последней переменной экземпляра, объявленной в классе аплета `RegPay`, является `nf`, которая представляет собой ссылку на объект типа `NumberFormat`, который будет описывать формат числа, используемого для вывода. Класс `NumberFormat` хранится в пакете `java.text`. Хотя формат выходных чисел можно выбрать и другими способами (например, с помощью класса `Formatter`), в дан-

ном случае удобно использовать именно класс NumberFormat, так как этот формат используется неоднократно и его можно установить один раз, в самом начале программы. Хороший пример его использования можно продемонстрировать именно с помощью финансовых аплетов.

Метод init()

Как и во всех аплетах, метод init() вызывается в начале выполнения аплета. Этот метод просто вызывает метод makeGUI() в потоке диспетчеризации событий. Как было сказано в главе 30, аплеты, основанные на библиотеке Swing, должны создаваться и взаимодействовать с компонентами GUI только при помощи потока диспетчеризации событий.

Метод makeGUI()

Метод makeGUI() устанавливает пользовательский интерфейс для аплета. Он выполняет следующее.

1. Заменяет диспетчер компоновки на диспетчер класса GridBagLayout.
2. Создает копии различных компонентов GUI.
3. Добавляет компоненты в сетку.
4. Добавляет слушатели событий компонентов.

Теперь давайте проанализируем строки метода makeGUI(). Он начинается следующими строками кода.

```
// Использование сеточной компоновки.
GridBagLayout gbag = new GridBagLayout();
GridBagConstraints gbc = new GridBagConstraints();
setLayout(gbag);
```

В этом фрагменте кода создается диспетчер компоновки класса GridBagLayout, который будет использоваться аплетом. (Более подробно о применении класса GridBagLayout рассказывалось в главе 25.) Использование диспетчера класса GridBagLayout объясняется тем, что с его помощью можно очень точно управлять размещением элементов управления в аплете.

Затем метод makeGUI() создает метки, текстовые поля и кнопку Compute, как показано ниже.

```
JLabel heading = new
    JLabel("Compute Monthly Loan Payments");

JLabel amountLab = new JLabel("Principal ");
JLabel periodLab = new JLabel("Years ");
JLabel rateLab = new JLabel("Interest Rate ");
JLabel paymentLab = new JLabel("Monthly Payments ");
amountText = new JTextField(10);
periodText = new JTextField(10);
paymentText = new JTextField(10);
rateText = new JTextField(10);

// Поле платежа только для отображения.
paymentText.setEditable(false);

doIt = new JButton("Compute");
```

Обратите внимание на то, что текстовому полю, отображающему ежемесячный платеж, устанавливается свойство, доступное только для чтения, при помощи вызова метода `setEditable(false)`. В результате поле окрашивается серым цветом и пользователь не может ввести в нем текст. Однако содержимое текстового поля можно установить с помощью вызова метода `setText()`. Таким образом, если в текстовом поле редактирование запрещено, то его можно использовать для отображения текста и пользователь не сможет его изменить.

В следующем фрагменте кода определяются ограничения сетки для каждого компонента.

```
// Определяем сетку.
gbc.weighty = 1.0; // используем строку, весом 1
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Располагаем большинство компонентов справа.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(amountLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(amountText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(paymentLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(paymentText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);
```

Хотя на первый взгляд этот фрагмент кода может показаться сложным, на самом деле это не так. Следует просто иметь в виду, что каждая строка в сетке определяется отдельно. Сейчас проанализируем работу этого фрагмента. Для начала вес каждой строки, содержащийся в переменной объекта `gbc.weighty`, получает значение `1`. Это позволит равномерно распределить дополнительное пространство в сетке там, где имеется больше интервалов между строками, чем это необходимо для хранения компонента. Затем переменной `gbc.gridwidth` присваивается значение `REMAINDER`, а переменной `gbc.anchor` — значение `NORTH`. Метка, на которую ссылается объект заголовка `heading`, добавляется в объект `gbag` при помощи вызова метода `setConstraints()`. В этом фрагменте устанавливается положение заголовка `heading` в верхней части сетки (значение `NORTH` — север), для него отводится оставшаяся часть строки. Таким образом, после выполнения этого фрагмента кода заголовок будет находиться в верхней части окна, оставаясь в то же время в строке.

Затем добавляются четыре текстовых поля и их метки. Сначала переменной gbc.anchor присваивается значение EAST. В результате каждый компонент будет выровнен вправо.

Далее переменной gbc.gridWidth присваивается значение RELATIVE и добавляется метка. После этого переменной gbc.gridWidth присваивается значение REMAINDER и добавляется текстовое поле. Таким образом, каждая пара текстового поля и метки будет занимать одну строку. Этот процесс повторяется до тех пор, пока не будут добавлены все четыре пары текстовых полей и меток. После всего этого кнопка Compute размещается в центре.

После того как будут определены ограничители сетки, следующий фрагмент кода добавляет в окно компоненты.

```
// Добавляем все компоненты.
add(heading);
add(amountLab);
add(amountText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(paymentLab);
add(paymentText);
add(doIt);
```

Затем регистрируются слушатели событий для трех текстовых полей ввода и кнопки Compute, как показано ниже.

```
// Регистрируем на получение событий действий.
amountText.addActionListener(this);
periodText.addActionListener(this);
rateText.addActionListener(this);
doIt.addActionListener(this);
```

В завершение получаем объект класса NumberFormat, а в качестве формата выбираются две десятичные цифры.

```
// Создаем формат числа.
nf = NumberFormat.getInstance();
nf.setMinimumFractionDigits(2);
nf.setMaximumFractionDigits(2);
```

При вызове фабричного метода getInstance() получаем объект класса NumberFormat, пригодный для размещения по умолчанию.

При вызове методов setMinimumFractionDigits() и setMaximumFractionDigits() определяется минимальное и максимальное количество отображаемых десятичных цифр. Так как минимальное и максимальное количество составляет 2, то видимыми всегда будут два десятичных знака.

Метод actionPerformed()

Метод actionPerformed() вызывается каждый раз при нажатии клавиши <Enter>, когда курсор находится в текстовом поле, или при щелчке на кнопке Compute. Этот метод выполняет три основные функции – получает введенную пользователем информацию о ссуде, вызывает метод compute() для поиска платежей по ссуде и отображает результат. Теперь давайте рассмотрим строки метода actionPerformed().

После объявления переменной `result` метод `paint()`, с помощью следующей последовательности кода, получает строки из трех полей, в которых пользователь вводит свои данные.

```
String amountStr = amountText.getText();
String periodStr = periodText.getText();
String rateStr = rateText.getText();
```

Затем начинается блок `try`, в котором проверяется, действительно ли все три поля содержат информацию.

```
try {
    if(amountStr.length() != 0 &&
       periodStr.length() != 0 &&
       rateStr.length() != 0) {
```

Напомним, что пользователь должен ввести исходную сумму выданной ссуды, срок погашения ссуды в годах и процент по ссуде. Если каждое из этих полей будет содержать информацию, то длина каждой строки будет больше нуля.

Если пользователь ввел все данные по ссуде, то числовые значения, соответствующие этим строкам, извлекаются и сохраняются в соответствующей переменной экземпляра. Затем вызывается метод `compute()` для расчета платежа по ссуде, и результат отображается в текстовом поле, доступном только для чтения, на которое ссылается объект `paymentText`, как показано ниже.

```
principal = Double.parseDouble(amountStr);
numYears = Double.parseDouble(periodStr);
intRate = Double.parseDouble(rateStr) / 100;

result = compute();

paymentText.setText(nf.format(result));
```

Обратите внимание на вызов метода `nf.format(result)`. Благодаря этому методу значение в переменной `result` будет иметь определенный ранее формат (с двумя десятичными цифрами) и будет возвращена результирующая строка. Эта строка впоследствии будет использоваться для установки текста в текстовом поле, определяемом с помощью объекта `paymentText`.

Если в одном из текстовых полей пользователь введет нечисловое значение, то метод `Double.parseDouble()` передает исключение `NumberFormatException`. Если это произойдет, в строке состояния будет отображено сообщение об ошибке и текстовое поле Payment (Платеж) окажется пустым, как показано ниже.

```
            showStatus(""); // удаляем любое предыдущее сообщение об ошибке
        } catch (NumberFormatException exc) {
            showStatus("Invalid Data");
            paymentText.setText("");
        }
}
```

В противном случае будет удалено любое отображенное ранее сообщение.

Метод `compute()`

Расчет платежа по ссуде производится с помощью метода `compute()`. Он реализует приведенную выше формулу и работает со значениями переменных `principal`, `intRate`, `numYears` и `payPerYear`. Метод возвращает результат расчета.

На заметку! Базовая структура, используемая в аплете `RegPay`, применяется во всех аплетах, представленных в этой главе.

Расчет будущей стоимости вклада

Еще один важный финансовый расчет позволяет определить будущую стоимость вклада на основании данных о первоначальной сумме вклада, норме прибыли, количестве периодов начисления сложного процента в течение одного года и количестве лет, на которые рассчитан вклад. Допустим, вы хотите узнать, сколько денег будет на вашем пенсионном счете через 12 лет, если на данный момент на нем имеется 98 000 долларов, а средняя годовая норма прибыли составляет 6 процентов. Для этих целей предназначен аплет `FutVal`.

Для расчета будущей стоимости вклада используется следующая формула.

```
Future Value = principal * ((rateOfRet / compPerYear) + 1) *
compPlerYear * numYears
```

Здесь *`rateOfRet`* определяет норму прибыли, *`principal`* — первоначальную сумму вклада, *`compPerYear`* — количество периодов начисления сложного процента в течение одного года, а *`numYears`* — срок вклада в годах. Если вы используете годовую норму прибыли для *`rateOfRet`*, то количество периодов начисления сложного процента в течение одного года составляет 1.

Аплет `FutVal` использует приведенную выше формулу для расчета будущей стоимости вклада. Аплет, созданный этой программой, показан на рис. 33.2. Помимо различий в расчетах в методе `compute()`, этот аплет похож на аплет `RegPay`, который мы рассматривали в предыдущем разделе.

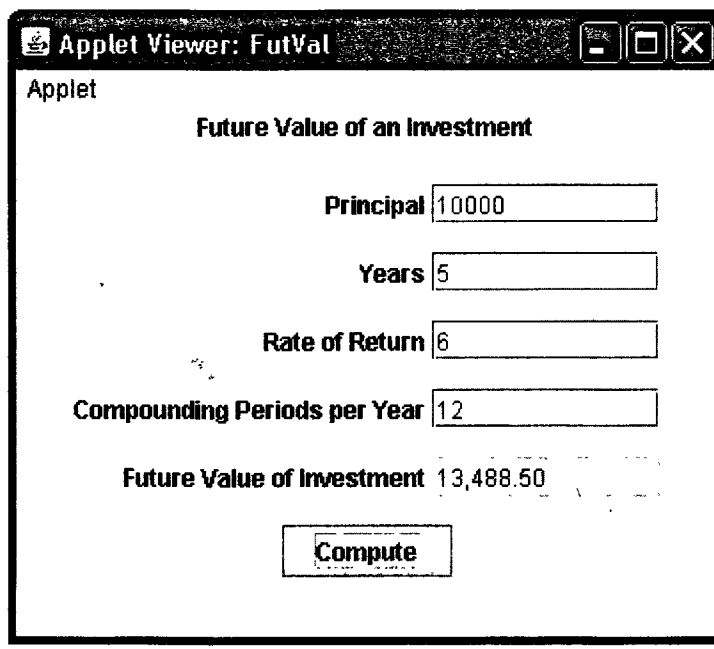


Рис. 33.2. Окно аплета `FutVal`

```
// Расчет первоначальной суммы вклада, необходимой для
// того, чтобы в будущем достичь требуемой стоимости вклада.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
import java.text.*;
/*
<applet code="FutVal" width=380 height=240>
</applet>
*/

public class FutVal extends JApplet
implements ActionListener {

    JTextField amountText, futvalText, periodText,
               rateText, compText;
    JButton doIt;

    double principal; // первоначальная стоимость
    double rateOfRet; // норма прибыли
    double numYears;  // срок вклада в годах
    int compPerYear;  // количество периодов начисления сложного
                      // процента в течение одного года
    NumberFormat nf;

    public void init() {
        try {
            SwingUtilities.invokeAndWait(new Runnable () {
                public void run() {
                    makeGUI(); // инициализация GUI
                }
            });
        } catch(Exception exc) {
            System.out.println("Can't create because of "+ exc);
         // System.out.println("Невозможно создать из-за "+ exc);
        }
    }

    // Устанавливаем и инициализируем GUI.
    private void makeGUI() {

        // Используем сеточную компоновку.
        GridBagLayout gbag = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(gbag);
        JLabel heading = new
            JLabel("Future Value of an Investment");
     // JLabel heading = new
     //                  JLabel("Расчет будущей стоимости вклада");
        JLabel amountLab = new JLabel("Principal ");
     // JLabel amountLab = new JLabel("Начальная сумма ");
        JLabel periodLab = new JLabel("Years ");
     // JLabel periodLab = new JLabel("Количество лет ");
        JLabel rateLab = new JLabel("Rate of Return ");
     // JLabel rateLab = new JLabel("Норма прибыли ");
        JLabel futvalLab = new
                           JLabel("Future Value of Investment ");
     // JLabel futvalLab = new
     //                    JLabel("Будущая стоимость вклада ");
        JLabel compLab = new
                         JLabel("Compounding Periods per Year ");
     // JLabel compLab = new
     //   JLabel("Количество периодов начисления сложного
     //             процента в год ");

        amountText = new JTextField(10);
        periodText = new JTextField(10);
        futvalText = new JTextField(10);
```

```
        rateText = new JTextField(10);
        compText = new JTextField(10);

        // Поле будущей стоимости, только для отображения.
        futvalText.setEditable(false);
        doIt = new JButton("Compute");
    // doIt = new JButton("Вычислить");

        // Определяем сетку.
        gbc.weighty = 1.0; // используем строку, весом 1
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbc.anchor = GridBagConstraints.NORTH;
        gbag.setConstraints(heading, gbc);

        // Размещаем большинство компонентов справа.
        gbc.anchor = GridBagConstraints.EAST;

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(amountLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(amountText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(periodLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(periodText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(rateLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(rateText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(compLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(compText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(futvalLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(futvalText, gbc);

        gbc.anchor = GridBagConstraints.CENTER;
        gbag.setConstraints(doIt, gbc);

        add(heading);
        add(amountLab);
        add(amountText);
        add(periodLab);
        add(periodText);
        add(rateLab);
        add(rateText);
        add(compLab);
        add(compText);
        add(futvalLab);
        add(futvalText);
        add(doIt);

        // Регистрируем на получение событий действий.
        amountText.addActionListener(this);
        periodText.addActionListener(this);
        rateText.addActionListener(this);
        compText.addActionListener(this);
```

```
        doIt.addActionListener(this);

        // Создаем формат числа.
        nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(2);
        nf.setMaximumFractionDigits(2);
    }

    /* Пользователь нажал <Enter> в текстовом поле или
       щелкнул на кнопке Compute. Отображаем результат,
      если все поля заполнены. */
    public void actionPerformed(ActionEvent ae) {
        double result = 0.0;

        String amountStr = amountText.getText();
        String periodStr = periodText.getText();
        String rateStr = rateText.getText();
        String compStr = compText.getText();

        try {
            if(amountStr.length() != 0 &&
               periodStr.length() != 0 &&
               rateStr.length() != 0 &&
               compStr.length() != 0) {

                principal = Double.parseDouble(amountStr);
                numYears = Double.parseDouble(periodStr);
                rateOfRet = Double.parseDouble(rateStr) / 100;
                compPerYear = Integer.parseInt(compStr);

                result = compute();

                futvalText.setText(nf.format(result));
            }
            showStatus(""); // удаляем любое предыдущее сообщение об
                            // ошибке
        } catch (NumberFormatException exc) {
            showStatus("Invalid Data");
         // showStatus("Неверные данные");
            futvalText.setText("");
        }
    }

    // Расчет будущей стоимости.
    double compute() {
        double b, e;

        b = (1 + rateOfRet/compPerYear);
        e = compPerYear * numYears;

        return principal * Math.pow(b, e);
    }
}
```

Расчет первоначальной суммы вклада, необходимой для достижения будущей суммы

Иногда необходимо узнать о том, сколько денег нужно положить в банк, чтобы в будущем достичь определенной суммы вклада. Например, если вы хотите с по-

мощью вклада заработать деньги на обучение своих детей и знаете, что для этого через 5 лет вам понадобится 75 000 долларов, то нужно рассчитать, сколько денег необходимо внести на счет сейчас, при условии что процентная ставка составляет 7% в год. Для этого предназначен аплет `InitInv`.

Формула для расчета первоначальной суммы вклада выглядит следующим образом.

```
Initial Investment=targetValue/ (((rateOfRet/compPerYear) + 1)
compPerYear*numYears)
```

Здесь *rateOfRet* определяет норму прибыли, *targetValue* — первоначальный баланс, *compPerYear* — количество периодов начисления сложного процента в течение одного года, а *numYears* — срок вклада. Если вы используете годовую норму прибыли для *rateOfRet*, то количество периодов начисления сложного процента в течение одного года будет равно 1.

Следующий аплет, `InitInv`, использует приведенную выше формулу для расчета первоначальной суммы вклада, необходимой для того, чтобы в будущем достичь требуемой суммы. Аплет, созданный этой программой, показан на рис. 33.3.

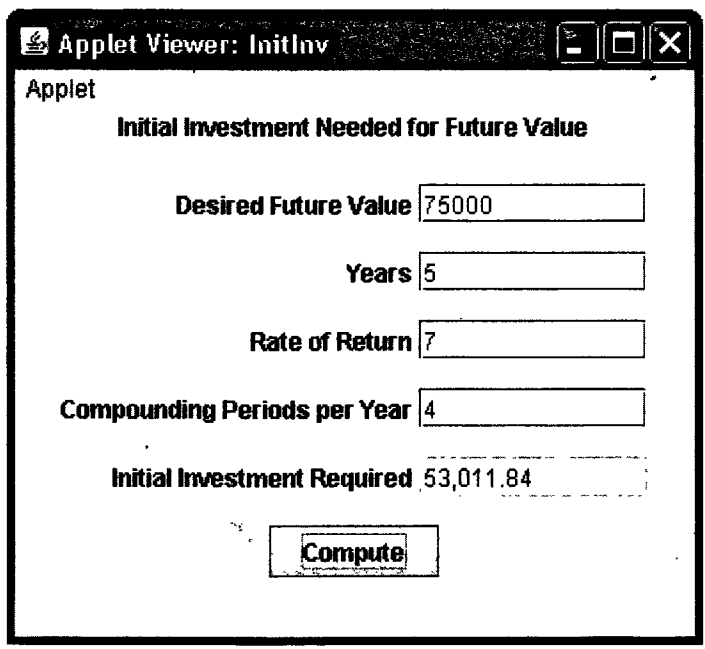


Рис. 33.3. Окно аплета `InitInv`

```
/* Расчет первоначальной суммы вклада, необходимой для
получения желаемого ежегодного дохода. Другими словами,
нужно рассчитать первоначальную сумму вклада, необходимую
для того чтобы регулярно получать доход в виде процентов
в течение определенного периода времени. */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;

/*
```

```
<applet code="InitInv" width=340 height=240>
</applet>
*/

public class InitInv extends JApplet
implements ActionListener {

    JTextField targetText, initialText, periodText,
               rateText, compText;
    JButton doIt;

    double targetValue; // первоначальное значение targetValue
    double rateOfRet;   // норма прибыли
    double numYears;    // срок вклада
    int compPerYear;    // количество периодов начисления сложного
                        // процента в течение одного года

    NumberFormat nf;
    public void init() {
        try {
            SwingUtilities.invokeAndWait(new Runnable () {
                public void run() {
                    makeGUI(); // инициализация GUI
                }
            });
        } catch(Exception exc) {
            System.out.println("Can't create because of "+ exc);
         // System.out.println("Невозможно создать из-за "+ exc);
        }
    }

    // Устанавливаем и инициализируем GUI.
    private void makeGUI() {
        // Используем сеточную компоновку.
        GridBagLayout gbag = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(gbag);
        JLabel heading = new
               JLabel("Initial Investment Needed for " +
                      "Future Value");
     // JLabel heading = new
     //        JLabel("Расчет первоначальной суммы вклада " +
     //               "для достижения будущей стоимости");
        JLabel targetLab = new JLabel("Desired Future Value ");
     // JLabel targetLab = new JLabel("Желаемая будущая сумма ");
        JLabel periodLab = new JLabel("Years ");
     // JLabel periodLab = new JLabel("Количество лет ");
        JLabel rateLab = new JLabel("Rate of Return ");
     // JLabel rateLab = new JLabel("Норма прибыли ");
        JLabel compLab = new
               JLabel("Compounding Periods per Year ");
     // JLabel compLab = new
     //        JLabel("Количество периодов начисления сложного процента
     //                в год ");
        JLabel initialLab = new
               JLabel("Initial Investment Required ");
     // JLabel initialLab = new
     //        JLabel("Требуемая первоначальная сумма вклада ");

        targetText = new JTextField(10);
        periodText = new JTextField(10);
```

```
        initialText = new JTextField(10);
        rateText = new JTextField(10);
        compText = new JTextField(10);

        // Поле исходного значения, только для отображения.
        initialText.setEditable(false);
        doIt = new JButton("Compute");
    // doIt = new JButton("Вычислить");

        // Определяем сетку.
        gbc.weighty = 1.0; // используем строку весом 1

        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbc.anchor = GridBagConstraints.NORTH;
        gbag.setConstraints(heading, gbc);

        // Размещаем большинство компонентов справа.
        gbc.anchor = GridBagConstraints.EAST;

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(targetLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(targetText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(periodLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(periodText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(rateLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(rateText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(compLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(compText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(initialLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(initialText, gbc);

        gbc.anchor = GridBagConstraints.CENTER;
        gbag.setConstraints(doIt, gbc);

        // Добавляем все компоненты.
        add(heading);
        add(targetLab);
        add(targetText);
        add(periodLab);
        add(periodText);
        add(rateLab);
        add(rateText);
        add(compLab);
        add(compText);
        add(initialLab);
        add(initialText);
        add(doIt);
```

```
        // Регистрируем на получение событий.
        targetText.addActionListener(this);
        periodText.addActionListener(this);
        rateText.addActionListener(this);
        compText.addActionListener(this);
        doIt.addActionListener(this);

        // Создаем формат числа.
        nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(2);
        nf.setMaximumFractionDigits(2);
    }

    /* Пользователь нажал <Enter> в текстовом поле или
       щелкнул на кнопке Compute. Отображаем результат,
       если заполнены все поля. */
    public void actionPerformed(ActionEvent ae) {
        double result = 0.0;

        String targetStr = targetText.getText();
        String periodStr = periodText.getText();
        String rateStr = rateText.getText();
        String compStr = compText.getText();

        try {
            if(targetStr.length() != 0 &&
               periodStr.length() != 0 &&
               rateStr.length() != 0 &&
               compStr.length() != 0) {

                targetValue = Double.parseDouble(targetStr);
                numYears = Double.parseDouble(periodStr);
                rateOfRet = Double.parseDouble(rateStr) / 100;
                compPerYear = Integer.parseInt(compStr);

                result = compute();

                initialText.setText(nf.format(result));
            }

            showStatus(""); // удаляем любое предыдущее сообщение
                            // об ошибке
        } catch (NumberFormatException exc) {
            showStatus("Invalid Data");
         // showStatus("Неверные данные");
            initialText.setText("");
        }
    }

    // Расчет требуемого первоначального вклада.
    double compute() {
        double b, e;

        b = (1 + rateOfRet/compPerYear);
        e = compPerYear * numYears;

        return targetValue / Math.pow(b, e);
    }
}
```

Расчет первоначальной суммы вклада, необходимой для получения желаемого годового дохода

Еще один распространенный финансовый расчет используется для определения денежной суммы, которую необходимо внести на счет, чтобы получать определенный годовой процент. Например, вы хотите после выхода на пенсию получать по 5 000 долларов каждый месяц в течение 20 лет. Вопрос заключается в том, сколько денег необходимо внести на счет сейчас, чтобы обеспечить будущие процентные выплаты? Ответ можно получить с помощью следующей формулы.

```
Initial Investment = ((regWD * wdPerYear) / rateOfRet) *
                     (1 - (1 / (rateOfRet / wdPerYear + 1)wdPerYear * numYears))
```

Здесь *rateOfRet* определяет норму прибыли, *regWD* — требуемый регулярный доход в виде процентов, *wdPerYear* — сколько раз в году вы будете получать доход в виде процентов, а *numYears* — в течение скольких лет вы планируете получать эти деньги.

Приведенный ниже код аплета `Annuity` рассчитывает сумму первоначального вклада, необходимую для получения желаемого ежегодного дохода. Аплет, созданный с помощью этой программы, показан на рис. 33.4.

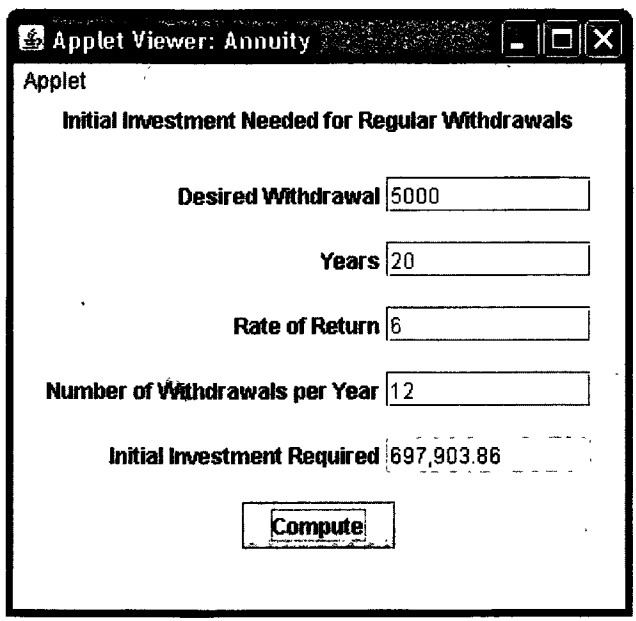


Рис. 33.4. Окно аплета `Annuity`

```
/* Расчет первоначальной суммы вклада, необходимой для
получения желаемого ежегодного дохода. Другими словами,
нужно рассчитать первоначальную сумму вклада, необходимую
для регулярного получения дохода в виде процентов в течение
определенного периода времени. */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
import java.text.*;
/*
<applet code="Annuity" width=340 height=260>
</applet>
*/

public class Annuity extends JApplet
implements ActionListener {

    JTextField regWDText, initialText, periodText,
               rateText, numWDText;
    JButton doIt;
    double regWDAmount; // доход, получаемый в виде процентов
    double rateOfRet;   // норма прибыли
    double numYears;    // срок вклада
    int numPerYear;     // сколько раз в году будет выплачиваться доход

    NumberFormat nf;

    public void init() {
        try {
            SwingUtilities.invokeAndWait(new Runnable () {
                public void run() {
                    makeGUI(); // инициализация GUI
                }
            });
        } catch(Exception exc) {
            System.out.println("Can't create because of "+ exc);
          // System.out.println("Невозможно создать из-за "+ exc);
        }
    }

    // Устанавливаем и инициализируем GUI.
    private void makeGUI() {

        // Использование сеточной компоновки.
        GridBagLayout gbag = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(gbag);
        JLabel heading = new
               JLabel("Initial Investment Needed for " +
                      "Regular Withdrawals");
     // JLabel heading = new
     //        JLabel("Расчет первоначальной суммы вклада для " +
     //               "желаемого ежегодного дохода");
        JLabel regWDLab = new JLabel("Desired Withdrawal ");
     // JLabel regWDLab = new JLabel("Желаемый ежегодный доход ");
        JLabel periodLab = new JLabel("Years ");
     // JLabel periodLab = new JLabel("Количество лет ");
        JLabel rateLab = new JLabel("Rate of Return ");
     // JLabel rateLab = new JLabel("Норма прибыли ");
        JLabel numWDLab = new
               JLabel("Number of Withdrawals per Year ");
     // JLabel numWDLab = new
     //        JLabel("Количество получений дохода в год ");
        JLabel initialLab = new
               JLabel("Initial Investment Required ");
     // JLabel initialLab = new
     //        JLabel("Требуемая первоначальная сумма вклада ");
        regWDText = new JTextField(10);

        periodText = new JTextField(10);
        initialText = new JTextField(10);
        rateText = new JTextField(10);
        numWDText = new JTextField(10);
```

```
// Поле первоначального вклада, только для отображения.
initialText.setEditable(false);
doIt = new JButton("Compute");
// doIt = new JButton("Вычислить");

// Определяем сетку.
gbc.weighty = 1.0; // используем строку, весом 1
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Размещаем большинство компонентов справа.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(regWDLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(regWDText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(numWDLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(numWDText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(initialLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(initialText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

// Добавляем все компоненты.
add(heading);
add(regWDLab);
add(regWDText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(numWDLab);
add(numWDText);
add(initialLab);
add(initialText);
add(doIt);

// Регистрируем на получение событий действий текстового поля.
regWDText.addActionListener(this);
periodText.addActionListener(this);
rateText.addActionListener(this);
numWDText.addActionListener(this);
doIt.addActionListener(this);

// Создание формата числа.
nf = NumberFormat.getInstance();
```

```
        nf.setMinimumFractionDigits(2);
        nf.setMaximumFractionDigits(2);
    }

    /* Пользователь нажал <Enter> в текстовом поле или
       щелкнул на кнопке Compute. Отображаем результат,
       если заполнены все поля. */
    public void actionPerformed(ActionEvent ae) {
        double result = 0.0;

        String regWDStr = regWDText.getText();
        String periodStr = periodText.getText();
        String rateStr = rateText.getText();
        String numWDStr = numWDText.getText();

        try {
        if(regWDStr.length() != 0 &&
           periodStr.length() != 0 &&
           rateStr.length() != 0 &&
           numWDStr.length() != 0) {

            regWDAmount = Double.parseDouble(regWDStr);
            numYears = Double.parseDouble(periodStr);
            rateOfRet = Double.parseDouble(rateStr) / 100;
            numPerYear = Integer.parseInt(numWDStr);

            result = compute();

            initialText.setText(nf.format(result));
        }
            showStatus(""); // удаляем любое предыдущее сообщение
                            // об ошибке
        } catch (NumberFormatException exc) {
            showStatus("Invalid Data");
         // showStatus("Неверные данные");
            initialText.setText("");
        }
    }

    // Расчет требуемой суммы первоначального вклада.
    double compute() {
        double b, e;
        double t1, t2;

        t1 = (regWDAmount * numPerYear) / rateOfRet;

        b = (1 + rateOfRet/numPerYear);
        e = numPerYear * numYears;

        t2 = 1 - (1 / Math.pow(b, e));
        return t1 * t2;
    }
}
```

Нахождение максимального годового дохода для данной суммы вклада

Этот расчет используется для определения максимального годового дохода (регулярные выплаты в виде процентов), который можно получать для данной суммы вклада в течение определенного периода времени. Например, если на ваш пенси-

онный счет внесено 500 000 долларов, то нужно узнать, сколько денег вы сможете получать из этой суммы ежемесячно в течение 20 лет, если процентная ставка составляет 6%. Формула для вычисления максимального дохода показана ниже.

```
Maximum Withdrawal = principal * (((rateOfRet / wdPerYear) /
        (-1 + ((rateOfRet / wdPerYear) + 1)wdPerYear * numYears)) +
        (rateOfRet / wdPerYear))
```

Здесь *rateOfRet* определяет норму прибыли, *principal* — сумму первоначального вклада, *wdPerYear* — сколько раз в году будет осуществляться выплата по процентам, а *numYears* — на какой срок (в годах) рассчитан вклад.

Приведенный ниже код аплета MaxWD рассчитывает максимальные периодические суммы дохода в виде процентов, которые можно получать в течение указанного периода времени с учетом определенной нормы прибыли. Аплет, созданный этой программой, показан на рис. 33.5.

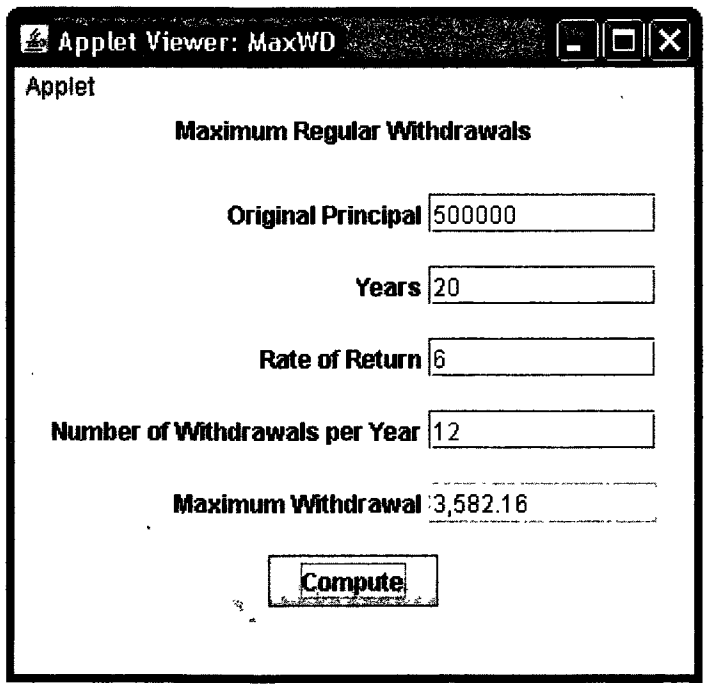


Рис. 33.5. Окно аплета MaxWD

```
/* Расчет максимального годового дохода, который можно получать
в виде процентов по вкладу в течение определенного
периода времени. */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
<applet code="MaxWD" width=340 height=260>
</applet>
*/
public class MaxWD extends JApplet
implements ActionListener {
    JTextField maxWDText, orgPText, periodText,
```

```
            rateText, numWDText;
JButton doIt;

double principal; // первоначальная сумма вклада
double rateOfRet; // годовая норма прибыли
double numYears;  // срок вклада в годах
int numPerYear;   // сколько раз в году будет производиться выплата

NumberFormat nf;

public void init() {
    try {
        SwingUtilities.invokeAndWait(new Runnable () {
            public void run() {
                makeGUI(); // инициализация GUI
            }
        });
    } catch(Exception exc) {
        System.out.println("Can't create because of "+ exc);
     // System.out.println("Невозможно создать из-за "+ exc);
    }
}

// Устанавливаем и инициализируем GUI.
private void makeGUI() {

    // Использование сеточной компоновки.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    JLabel heading = new
           JLabel("Maximum Regular Withdrawals");
 // JLabel heading = new
 // JLabel("Расчет максимального годового дохода");
    JLabel orgPLab = new JLabel("Original Principal ");
 // JLabel orgPLab = new JLabel("Первоначальная сумма вклада ");
    JLabel periodLab = new JLabel("Years ");
 // JLabel periodLab = new JLabel("Количество лет ");
    JLabel rateLab = new JLabel("Rate of Return ");
 // JLabel rateLab = new JLabel("Норма прибыли ");
    JLabel numWDLab =
    new JLabel("Number of Withdrawals per Year ");
 // JLabel numWDLab =
 // new JLabel("Количество получений дохода в год ");
    JLabel maxWDLab = new JLabel("Maximum Withdrawal ");
 // JLabel maxWDLab = new JLabel("Максимальный годовой доход ");

    maxWDText = new JTextField(10);
    periodText = new JTextField(10);
    orgPText = new JTextField(10);
    rateText = new JTextField(10);
    numWDText = new JTextField(10);

    // Поле максимальной суммы дохода, только для отображения.
    maxWDText.setEditable(false);

    doIt = new JButton("Compute");
 // doIt = new JButton("Вычислить");

    // Определяем сетку.
    gbc.weighty = 1.0; // используем строку, весом 1
```

```
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbc.anchor = GridBagConstraints.NORTH;
        gbag.setConstraints(heading, gbc);

        // Размещаем большинство компонентов справа.
        gbc.anchor = GridBagConstraints.EAST;

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(orgPLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(orgPText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(periodLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(periodText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(rateLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(rateText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(numWDLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(numWDText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(maxWDLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(maxWDText, gbc);

        gbc.anchor = GridBagConstraints.CENTER;
        gbag.setConstraints(doIt, gbc);

        // Добавляем все компоненты.
        add(heading);
        add(orgPLab);
        add(orgPText);
        add(periodLab);
        add(periodText);
        add(rateLab);
        add(rateText);
        add(numWDLab);
        add(numWDText);
        add(maxWDLab);
        add(maxWDText);
        add(doIt);

        // Регистрируем на получение событий действий.
        orgPText.addActionListener(this);
        periodText.addActionListener(this);
        rateText.addActionListener(this);
        numWDText.addActionListener(this);
        doIt.addActionListener(this);

        // Создаем числовой формат.
        nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(2);
        nf.setMaximumFractionDigits(2);
    }

    /* Пользователь нажал <Enter> в текстовом поле или
```

```
    щелкнул на кнопке Compute. Отображаем результат,
    если все поля заполнены. */
    public void actionPerformed(ActionEvent ae) {
        double result = 0.0;

        String orgPStr = orgPText.getText();
        String periodStr = periodText.getText();
        String rateStr = rateText.getText();
        String numWDStr = numWDText.getText();

        try {
            if(orgPStr.length() != 0 &&
               periodStr.length() != 0 &&
               rateStr.length() != 0 &&
               numWDStr.length() != 0) {

                principal = Double.parseDouble(orgPStr);
                numYears = Double.parseDouble(periodStr);
                rateOfRet = Double.parseDouble(rateStr) / 100;
                numPerYear = Integer.parseInt(numWDStr);

                result = compute();

                maxWDText.setText(nf.format(result));
            }

            showStatus(""); // удаляем любое предыдущее сообщение
                            // об ошибке
        } catch (NumberFormatException exc) {
            showStatus("Invalid Data");
         // showStatus("Неверные данные");
            maxWDText.setText("");
        }
    }

    // Расчет максимальной регулярной суммы дохода.
    double compute() {
        double b, e;
        double t1, t2;

        t1 = rateOfRet / numPerYear;

        b = (1 + t1);
        e = numPerYear * numYears;

        t2 = Math.pow(b, e) - 1;

        return principal * (t1/t2 + t1);
    }
}
```

Нахождение остатка баланса по ссуде

Нередко бывает необходимо узнать остаток баланса по ссуде. Рассчитать его можно очень просто, если знать первоначальную сумму, процентную ставку и количество произведенных платежей. Чтобы найти остаток баланса, необходимо сложить платежи, вычитая из каждого платежа сумму, выделенную под проценты, и полученный результат вычесть из исходной суммы.

Приведенный ниже код аплета `RemBal` находит остаток баланса по ссуде. Аплет, созданный этой программой, показан на рис. 33.6.

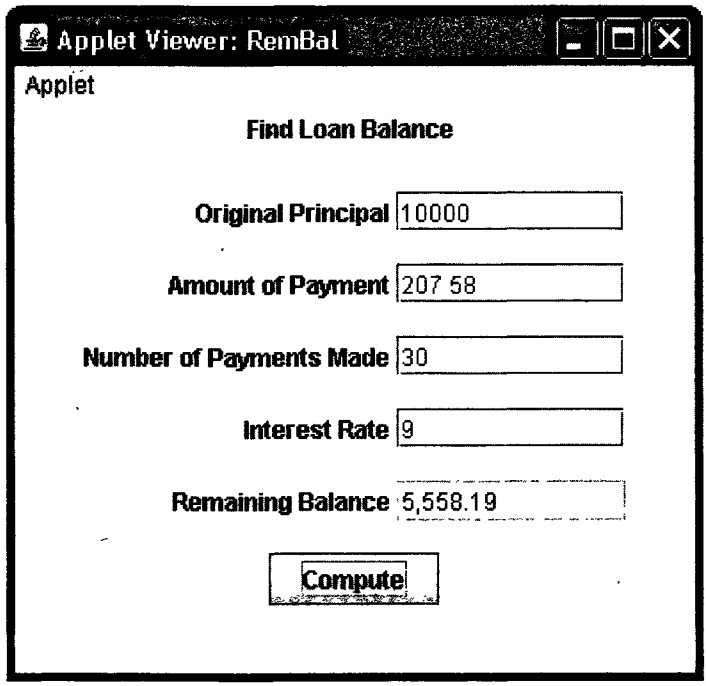


Рис. 33.6. Окно аплета `RemBal`

```
// Нахождение остатка баланса по ссуде.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
<applet code="RemBal" width=340 height=260>
</applet>
*/

public class RemBal extends JApplet
implements ActionListener {
    JTextField orgPText, paymentText, remBalText,
               rateText, numPayText;
    JButton doIt;

    double orgPrincipal; // исходная сумма
    double intRate; // процент по ссуде
    double payment; // сумма каждого платежа
    double numPayments; // количество произведенных платежей

    /* Количество платежей в течение одного года. Можно сделать
    так, чтобы это значение вводил пользователь. */
    final int payPerYear = 12;

    NumberFormat nf;

    public void init() {
        try {
            SwingUtilities.invokeAndWait(new Runnable () {
```

```
            public void run() {
                makeGUI(); // инициализация GUI
            }
        });
    } catch(Exception exc) {
        System.out.println("Can't create because of "+ exc);
     // System.out.println("Невозможно создать из-за "+ exc);
    }
}

// Устанавливаем и инициализируем GUI.
private void makeGUI() {
    // Используем сеточную компоновку.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    JLabel heading = new
           JLabel("Find Loan Balance ");
 // JLabel heading = new
 //        JLabel("Find Loan Balance ");
    JLabel orgPLab = new JLabel("Original Principal ");
 // JLabel orgPLab = new JLabel("Первоначальная сумма вклада ");
    JLabel paymentLab = new JLabel("Amount of Payment ");
 // JLabel paymentLab = new JLabel("Сумма платежа ");
    JLabel numPayLab = new JLabel("Number of Payments Made ");
 // JLabel numPayLab = new JLabel("Количество сделанных платежей ");
    JLabel rateLab = new JLabel("Interest Rate ");
 // JLabel rateLab = new JLabel("Процент по ссуде ");
    JLabel remBalLab = new JLabel("Remaining Balance ");
 // JLabel remBalLab = new JLabel("Остаточный баланс ");
    orgPText = new JTextField(10);
    paymentText = new JTextField(10);
    remBalText = new JTextField(10);
    rateText = new JTextField(10);
    numPayText = new JTextField(10);

    // Поле платежей, только для отображения.
    remBalText.setEditable(false);
    doIt = new JButton("Compute");
 // doIt = new JButton("Вычислить");

    // Определяем сетку.
    gbc.weighty = 1.0; // используем строку, весом 1

    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbc.anchor = GridBagConstraints.NORTH;
    gbag.setConstraints(heading, gbc);

    // Размещаем большинство компонентов справа.
    gbc.anchor = GridBagConstraints.EAST;

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(orgPLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(orgPText, gbc);

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(paymentLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(paymentText, gbc);

    gbc.gridwidth = GridBagConstraints.RELATIVE;
```

```
        gbag.setConstraints(rateLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(rateText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(numPayLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(numPayText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(remBalLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(remBalText, gbc);

        gbc.anchor = GridBagConstraints.CENTER;
        gbag.setConstraints(doIt, gbc);

        // Добавляем все компоненты.
        add(heading);
        add(orgPLab);
        add(orgPText);
        add(paymentLab);
        add(paymentText);
        add(numPayLab);
        add(numPayText);
        add(rateLab);
        add(rateText);
        add(remBalLab);
        add(remBalText);
        add(doIt);

        // Регистрируем на получение событий действия.
        orgPText.addActionListener(this);
        numPayText.addActionListener(this);
        rateText.addActionListener(this);
        paymentText.addActionListener(this);
        doIt.addActionListener(this);

        // Создаем числовой формат.
        nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(2);
        nf.setMaximumFractionDigits(2);
    }
    /* Пользователь нажал <Enter> в текстовом поле
    или щелкнул на кнопке Compute. Отображаем результат,
    если заполнены все поля. */
    public void actionPerformed(ActionEvent ae) {
        double result = 0.0;

        String orgPStr = orgPText.getText();
        String numPayStr = numPayText.getText();
        String rateStr = rateText.getText();
        String payStr = paymentText.getText();

        try {
            if(orgPStr.length() != 0 &&
               numPayStr.length() != 0 &&
               rateStr.length() != 0 &&
               payStr.length() != 0) {

                orgPrincipal = Double.parseDouble(orgPStr);
                numPayments = Double.parseDouble(numPayStr);
                intRate = Double.parseDouble(rateStr) / 100;
```

```
                payment = Double.parseDouble(payStr);

                result = compute();

                remBalText.setText(nf.format(result));
            }
            showStatus(""); // удаляем любое предыдущее сообщение
                            // об ошибке
        } catch (NumberFormatException exc) {
            showStatus("Invalid Data");
         // showStatus("Неверные данные");
            remBalText.setText("");
        }
    }

    // Расчет баланса по ссуде.
    double compute() {

        double bal = orgPrincipal;
        double rate = intRate / payPerYear;

        for(int i = 0; i < numPayments; i++)
            bal -= payment - (bal * rate);

        return bal;
    }
}
```

Создание финансовых сервлетов

Несмотря на простоту создания и использования аплетов, они составляют всего лишь половину "уравнения" Java для Интернета. Другая половина остается за сервлетами. Во время соединения сервлеты выполняются на стороне сервера, и для некоторых приложений более подходящим вариантом являются именно сервлеты. Поскольку многим читателям может понадобиться использовать в своих коммерческих приложениях сервлеты вместо аплетов, далее в этой главе будет показан способ преобразования финансовых аплетов в сервлеты.

Поскольку все финансовые аплеты построены на одной и той же базовой структуре, для преобразования выберем один аплет — `RegPay`. Предложенный базовый процесс вы сможете применять для преобразования любого аплета в сервлет. Вы сможете убедиться в том, что сделать это совсем не трудно.

На заметку! Сведения по созданию, проверке и запуску сервлетов можно получить в главе 32.

Преобразование аплета `RegPay` в сервлет

Преобразовать аплет расчета ссуды `RegPay` в сервлет несложно. Для начала сервлет должен импортировать пакеты `javax.servlet` и `javax.servlet.http`. Он должен также расширять класс `HttpServlet`, а не класс `JApplet`. Кроме этого, потребуется удалить весь код GUI. Затем необходимо добавить код, который получает параметры, передаваемые сервлету кодом HTML, который будет вызывать сервлет. Также сервлет должен посылать код HTML, отображающий результаты. Базовые финансовые расчеты остаются теми же. Изменяется лишь способ получения данных и их отображение.

Сервлет RegPayS

Следующий класс RegPayS является серверной версией (сервлетом) аплета RegPay. Предполагается, что файл RegPayS.class будет храниться в каталоге примеров сервлетов контейнера Tomcat (см. главу 32). Не забудьте ввести его имя в файл web.xml (также см. главу 32).

```
// Простой сервлет для расчета ссуды.
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.text.*;

public class RegPayS extends HttpServlet {
    double principal; // исходная сумма
    double intRate;   // процент по ссуде
    double numYears;  // срок, на который выдана ссуда

    /* Количество платежей в году. Можно сделать так,
       чтобы это значение вводил пользователь. */
    final int payPerYear = 12;

    NumberFormat nf;

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
    throws ServletException, IOException {
        String payStr = "";

        // Создаем числовой формат.
        nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(2);
        nf.setMaximumFractionDigits(2);

        // Получаем параметры.
        String amountStr = request.getParameter("amount");
        String periodStr = request.getParameter("period");
        String rateStr = request.getParameter("rate");

        try {
            if(amountStr != null && periodStr != null &&
               rateStr != null) {
                principal = Double.parseDouble(amountStr);
                numYears = Double.parseDouble(periodStr);
                intRate = Double.parseDouble(rateStr) / 100;
                payStr = nf.format(compute());
            }
            else { // пропущен один или более параметров
                amountStr = "";

                periodStr = "";
                rateStr = "";
            }
        } catch (NumberFormatException exc) {
            // Предпринимаем соответствующее действие.
        }

        // Задаем тип содержимого.
        response.setContentType("text/html");

        // Получаем поток выходных данных.
        PrintWriter pw = response.getWriter();
```

```
        // Отображаем необходимый код HTML.
        // Введите начальный баланс
        pw.print("<html><body> <left>" +
                 "<form name=\"Form1\"" +
                 " action=\"http://localhost:8080/" +
                 "examples/servlets/servlet/RegPayS\">" +
                 "<B>Enter amount to finance:</B>" +
                 " <input type=textbox name=\"amount\"" +
                 " size=12 value=\"");
        pw.print(amountStr + "\">");
        // Введите количество лет
        pw.print("<BR><B>Enter term in years:</B>" +
                 " <input type=textbox name=\"period\""+
                 " size=12 value=\"");
        pw.println(periodStr + "\">");
        // Введите процент по ссуде
        pw.print("<BR><B>Enter interest rate:</B>" +
                 " <input type=textbox name=\"rate\"" +
                 " size=12 value=\"");
        pw.print(rateStr + "\">");
        // Ежемесячный платеж
        pw.print("<BR><B>Monthly Payment:</B>" +
                 " <input READONLY type=textbox" +
                 " name=\"payment\" size=12 value=\"");
        pw.print(payStr + "\">");
        // Отправить
        pw.print("<BR><P><input type=submit value=\"Submit\">");
        pw.println("</form> </body> </html>");
    }

    // Расчет платежа по ссуде.
    double compute() {
        double numer;
        double denom;
        double b, e;

        numer = intRate * principal / payPerYear;
        e = -(payPerYear * numYears);
        b = (intRate / payPerYear) + 1.0;

        denom = 1.0 - Math.pow(b, e);

        return numer / denom;
    }
}
```

Первое, на что следует обратить внимание в сервлете `RegPayS`, — это то, что он имеет только два метода, `doGet()` и `compute()`. Метод `compute()` такой же, как и в аплете. Метод `doGet()` определяется в классе `HttpServlet`, который расширяет класс `RegPayS`. Этот метод вызывается сервером, когда сервлет должен ответить на запрос `GET`. Заметьте, что он передает ссылку объектам интерфейсов `HttpServletRequest` и `HttpServletResponse`, связанным с запросом.

Из параметра `request` сервлет получает аргументы, связанные с запросом. Для этого он вызывает метод `getParameter()`. Параметр возвращается в строковом виде. Поэтому числовое значение необходимо вручную преобразовать в двоичный формат. Если ни один из параметров не доступен, возвращается значение `null`.

Из объекта `response` сервлет получает поток, в который можно записать информацию ответа. Затем ответ возвращается браузеру при помощи вывода в данный поток. Прежде чем получить объект класса `PrintWriter` в потоке ответа, в качестве типа вывода необходимо определить `text/html`, вызвав для этого метод `setContentType()`.

Сервлет RegPayS можно вызывать как с параметрами, так и без них. При вызове без параметров сервлет отвечает кодом HTML, отображающим пустую форму калькулятора ссуд. В противном случае при вызове со всеми необходимыми параметрами сервлет RegPayS рассчитывает платеж по ссуде и повторно отображает форму, на этот раз с заполненным полем платежа.

Вызвать сервлет RegPayS проще всего, если связаться с его адресом URL, не передавая никаких параметров. Например, если предположить, что вы используете контейнер Tomcat, можно использовать следующую строку.

```
<A HREF = "http://localhost:8080/examples/servlets/servlet/
RegPayS">Калькулятор ссуд</A>
```

В результате будет отображена ссылка "Калькулятор ссуд" на сервлет RegPayS, расположенный в каталоге примеров сервлетов контейнера Tomcat. Обратите внимание на то, что при этом никакие параметры не передаются. Сервлет RegPayS будет возвращать полный код HTML, отображающий пустую страницу калькулятора ссуд.

Сервлет RegPayS можно вызвать и по-другому, если сначала отобразить вручную пустую форму. Этот прием показан ниже, и тоже с использованием каталога примеров сервлетов контейнера Tomcat.

```
<html>
<body>
<form name="Form1"
    action="http://localhost:8080/examples/servlets/servlet/RegPayS">
<B>Enter amount to finance:</B>
<input type=textbox name="amount" size=12 value="">
<BR>
<B>Enter term in years:</B>
<input type=textbox name="period" size=12 value="">
<BR>
<B>Enter interest rate:</B>
<input type=textbox name="rate" size=12 value="">
<BR>
<B>Monthly Payment:</B>
<input READONLY type=textbox name="payment"
    size=12 value="">
<BR><P>
<input type=submit value="Submit">
</form>
</body>
</html>
```

Самостоятельная работа

Первое, что вы можете попробовать сделать, — преобразовать другие финансовые аплеты в сервлеты. Поскольку все финансовые аплеты имеют одну и ту же структуру, достаточно будет повторить те же действия, что и для сервлета RegPayS. Существует множество других финансовых расчетов, которые вы найдете полезными для реализации в качестве аплетов или сервлетов (например, расчет нормы прибыли по вкладу или расчет суммы обычного вклада, которая необходима для того, чтобы в будущем достичь определенной суммы). Можно также вывести график погашения ссуды. Попытайтесь создать крупное приложение для выполнения расчетов, описанных в этой главе, позволяя пользователю выбирать необходимый расчет из меню.

ГЛАВА

34 Создание утилиты загрузки на языке Java

Сталкивались ли вы когда-нибудь с внезапным прекращением загрузки данных из Интернета, в результате чего приходилось начинать загрузку сначала? Если же вы подключены к Интернету при помощи коммутируемого соединения, то, скорее всего, вам приходилось переживать подобные досадные моменты не раз. Прекращение процесса загрузки может быть вызвано чем угодно, начиная с отключений во время ожидания звонка и заканчивая сбоем самой операционной системы. Но даже при высокоскоростном соединении нарушения связи все еще вполне возможны. Если начинать процесс загрузки снова и снова, для этого понадобится очень много времени и нервов, что в итоге может привести вас в полное уныние.

Пользователи нередко забывают о том факте, что во многих случаях при внезапном прекращении процесса загрузки его можно возобновить с того места, в каком он был прекращен, не начиная все с самого начала. В этой главе займемся созданием инструментального средства Download Manager (Диспетчер загрузки), которое будет управлять процессами загрузки данных из Интернета и позволит возобновлять прерванную загрузку данных. Это средство позволит приостанавливать и продолжать загрузку данных, а также управлять несколькими процессами загрузки одновременно.

Самым главным достоинством утилиты Download Manager является то, что с ее помощью можно загружать только определенные порции файла. В классическом сценарии загружается весь файл целиком. Если передача файла по какой-либо причине будет внезапно прекращена, то попытка завершить его загрузку не увенчается успехом. А утилита Download Manager может возобновить загрузку файла с того места, в котором произошел сбой, и загрузить оставшийся фрагмент файла. Следует учесть, однако, что не все процессы загрузки создаются одинаковыми, и некоторые из них вообще невозможно возобновить после внезапного прекращения. В следующем разделе вы узнаете о том, для каких файлов можно возобновлять процесс загрузки, а для каких — нет.

Инструмент Download Manager — это не только полезная утилита, но и превосходный пример мощности и лаконичности встроенных API Java, особенно в случае работы с Интернетом. Поскольку создатели Java заботились, прежде всего, о возможностях работы с Интернетом, неудивительно, что сетевые возможности Java являются непревзойденными. Например, если вы попытаетесь создать утилиту загрузки на другом языке программирования (например, C++), то для этого вам придется приложить гораздо больше усилий и вы обязательно столкнетесь с рядом трудностей.

Загрузка данных из Интернета

Чтобы разобраться с тем, как работает утилита Download Manager, необходимо выяснить, как на самом деле производится загрузка данных из Интернета.

Процесс загрузки данных из Интернета в самом простом виде представляет собой обыкновенные транзакции между клиентом и сервером. Клиент, в качестве которого выступает ваш браузер, просит загрузить файл, хранящийся на сервере в Интернете. Сервер отвечает ему, отправляя запрошенный файл браузеру. Чтобы клиенты могли взаимодействовать с серверами, они должны иметь установленный протокол. Самыми распространенными протоколами для загрузки файлов являются FTP (File Transfer Protocol – протокол передачи файлов) и HTTP (Hypertext Transfer Protocol – протокол передачи гипертекста). Протокол FTP обычно используется для обмена файлами между компьютерами, а протокол HTTP – для передачи веб-страниц и связанных с ними файлов (графических изображений, звуковых файлов и т.п.). Со временем, когда система World Wide Web приобрела большую популярность, протокол HTTP стал доминирующим для загрузки файлов из Интернета. Это, однако, не означает, что роль протокола FTP перестала быть важной.

Утилита Download Manager, созданием которой займемся в этой главе, будет поддерживать загрузку только по протоколу HTTP; это ограничение продиктовано лишь краткостью изложения материала по данной теме. Однако если вы добавите поддержку протокола FTP, это послужит вам хорошим уроком по расширению кода. Процесс загрузки с помощью протокола HTTP может происходить в двух вариантах – с возможностью возобновления (HTTP 1.1) и без возможности возобновления (HTTP 1.0). Они отличаются между собой способом запроса файлов, хранящихся на серверах. Если используется устаревший вариант протокола HTTP 1.0, то клиент может выслать серверу запрос, чтобы он отправил ему файл, тогда как с помощью протокола HTTP 1.1 клиент может попросить сервер выслать как весь файл целиком, так и *только определенную его порцию*. Именно на этой особенности и будет построена наша утилита Download Manager.

Обзор утилиты Download Manager

Утилита Download Manager использует простой, но тем не менее эффективный графический интерфейс, построенный с помощью библиотеки Java Swing. Окно утилиты Download Manager показано на рис. 34.1. Использование библиотеки Swing позволяет придать интерфейсу живой и современный внешний вид.

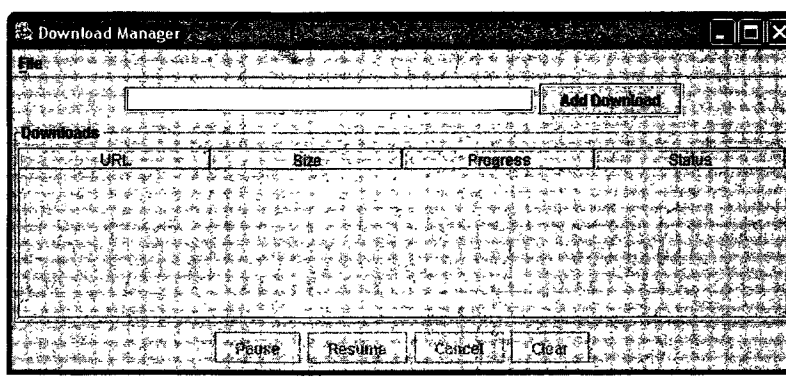


Рис. 34.1. Графический интерфейс утилиты Download Manager

Графический интерфейс поддерживает список процессов загрузки, управляемых в данный момент времени. Для каждого процесса загрузки в списке указывается его адрес URL, размер файла в байтах, количество загруженных данных

(в процентах) и текущее состояние. Процесс загрузки может иметь одно из следующих состояний: Downloading (обозначает непосредственно сам процесс загрузки), Paused (процесс загрузки приостановлен), Complete (процесс загрузки завершен), Error (ошибка процесса загрузки) и Cancelled (процесс загрузки отменен). Графический интерфейс позволяет также добавлять процессы загрузки в список и изменять состояние каждого процесса, присутствующего в списке. Если в списке выбрать какой-нибудь процесс загрузки, то, в зависимости от текущего состояния, его можно приостановить, возобновить, отменить или вообще удалить из списка.

Утилита Download Manager содержит несколько классов, позволяющих естественным образом разделить функциональные компоненты. Это классы `Download`, `DownloadsTableModel`, `ProgressRenderer` и `DownloadManager`.

Класс `DownloadManager` отвечает за графический пользовательский интерфейс и использует классы `DownloadsTableModel` и `ProgressRenderer` для отображения текущего списка процессов загрузки. Класс `Download` представляет "управляемый" процесс загрузки и отвечает за непосредственную загрузку файла. В следующих разделах рассмотрим каждый из этих классов подробно, уделяя особое внимание их внутренней работе и объясняя, как они связаны друг с другом.

Класс `Download`

Класс `Download` является главным классом утилиты Download Manager (или ее "рабочей лошадкой"). Его основная задача заключается в загрузке файла и сохранении его содержимого на диске. При каждом добавлении нового процесса загрузки в утилиту Download Manager создается новый объект класса `Download`, который и будет заниматься обслуживанием данного процесса.

Утилита Download Manager может загружать несколько файлов одновременно. Для этого необходимо, чтобы каждый процесс выполнялся независимо от остальных процессов. Кроме того, нужна возможность управлять состоянием каждого отдельного процесса, что позволит отображать его в интерфейсе. Это можно сделать с помощью класса `Download`.

Ниже представлен полный код класса `Download`. Обратите внимание на то, что он расширяет класс `Observable` и реализует интерфейс `Runnable`. Каждую часть проанализируем детально в следующих разделах.

```
import java.io.*;
import java.net.*;
import java.util.*;

// Этот класс загружает файл, на который указывает адрес URL.
class Download extends Observable implements Runnable {

    // Максимальный размер буфера загрузки.
    private static final int MAX_BUFFER_SIZE = 1024;

    // Названия состояний.
    public static final String STATUSES[] = {"Downloading",
                        "Paused", "Complete", "Cancelled", "Error"};

    // Это коды состояний.
    public static final int DOWNLOADING = 0;
    public static final int PAUSED = 1;
    public static final int COMPLETE = 2;
    public static final int CANCELLED = 3;
    public static final int ERROR = 4;
    private URL url;          // загрузка адреса URL
```

```
    private int size;       // размер загружаемых данных в байтах
    private int downloaded; // количество загруженных байтов
    private int status;     // текущее состояние процесса загрузки

    // Конструктор для класса Download.
    public Download(URL url) {
        this.url = url;
        size = -1;
        downloaded = 0;
        status = DOWNLOADING;

        // Начало процесса загрузки.
        download();
    }

    // Получаем адрес URL для данного процесса загрузки.
    public String getUrl() {
        return url.toString();
    }

    // Определяем размер загружаемых данных.
    public int getSize() {
        return size;
    }

    // Получаем информацию о ходе данного процесса загрузки.
    public float getProgress() {
        return ((float) downloaded / size) * 100;
    }

    // Получаем сведения о состоянии данного процесса загрузки.
    public int getStatus() {
        return status;
    }

    // Приостанавливаем данный процесс загрузки.
    public void pause() {
        status = PAUSED;
        stateChanged();
    }

    // Возобновляем данный процесс загрузки.
    public void resume() {
        status = DOWNLOADING;
        stateChanged();
        download();
    }

    // Отменяем данный процесс загрузки.
    public void cancel() {
        status = CANCELLED;
        stateChanged();
    }

    // В процессе загрузки возникла ошибка.
    private void error() {
        status = ERROR;
        stateChanged();
    }

    // Начинаем или возобновляем процесс загрузки.
    private void download() {
        Thread thread = new Thread(this);
```

```
        thread.start();
    }

    // Извлекаем имя файла из адреса URL.
    private String getFileName(URL url) {
        String fileName = url.getFile();
        return fileName.substring(fileName.lastIndexOf('/') + 1);
    }

    // Загружаем файл.
    public void run() {
        RandomAccessFile file = null;
        InputStream stream = null;

        try {
            // Открываем соединение с данным адресом URL.
            HttpURLConnection connection =
                    (HttpURLConnection) url.openConnection();

            // Определяем, какую часть файла нужно загрузить.
            connection.setRequestProperty("Range",
                                          "bytes=" + downloaded + "-");

            // Выполняем соединение с сервером.
            connection.connect();

            // Убеждаемся в том, что код ответа находится в
            // диапазоне 200.
            if (connection.getResponseCode() / 100 != 2) {
                error();
            }

            // Проверяем, имеет ли содержимое допустимую длину.
            int contentLength = connection.getContentLength();
            if (contentLength < 1) {
                error();
            }

            /* Задаем размер для данного процесса загрузки,
               если он еще не был задан. */
            if (size == -1) {
                size = contentLength;
                stateChanged();
            }

            // Открываем файл и ищем конец файла.
            file = new RandomAccessFile(getFileName(url), "rw");
            file.seek(downloaded);

            stream = connection.getInputStream();
            while (status == DOWNLOADING) {
                /* Задаем размер буфера так, чтобы загрузить
                   оставшуюся часть файла. */
                byte buffer[];
                if (size - downloaded > MAX_BUFFER_SIZE) {
                    buffer = new byte[MAX_BUFFER_SIZE];
                } else {
                    buffer = new byte[size - downloaded];
                }

                // Производим чтение из сервера в буфер.
                int read = stream.read(buffer);
                if (read == -1)
                    break;
```

```
                // Записываем содержимое буфера в файл.
                file.write(buffer, 0, read);
                downloaded += read;
                stateChanged();
            }

            /* Определяем состояние как завершенное, если была
               достигнута эта точка, поскольку в ней загрузка завершена.
             */
            if (status == DOWNLOADING) {
                status = COMPLETE;
                stateChanged();
            }
        } catch (Exception e) {
            error();
        } finally {

            // Закрываем файл.
            if (file != null) {
                try {
                    file.close();
                } catch (Exception e) {}
            }

            // Закрываем соединение с сервером.
            if (stream != null) {
                try {
                    stream.close();
                } catch (Exception e) {}
            }
        }
    }

    // Уведомляем наблюдателей об изменении состояния данного
    // процесса загрузки.
    private void stateChanged() {
        setChanged();
        notifyObservers();
    }
}
```

Переменные класса `Download`

Класс `Download` начинается с объявления нескольких статических финальных переменных, определяющих различные константы, используемые в классе. Затем объявляются четыре переменные экземпляра. Переменная `url` хранит адрес URL файла, который необходимо загрузить, переменная `size` — размер этого файла в байтах, переменная `download` — количество байтов, загруженных на данный момент, а переменная `status` показывает текущее состояние процесса загрузки.

Конструктор класса `Download`

Конструктор класса `Download` передает ссылку на адрес URL для загрузки в виде объекта класса `URL`, который присваивается переменной экземпляра `url`. Затем он присваивает остальным переменным экземпляра их исходные значения и вызывает метод `download()`. Обратите внимание на то, что переменная `size` имеет значение `-1`, которое показывает, что размер еще не определен.

Метод download()

Метод download() создает новый объект класса Thread, передавая ему ссылку для вызова экземпляра класса Download. Как уже было сказано ранее, необходимо, чтобы каждый процесс загрузки выполнялся отдельно от остальных. Чтобы объект класса Download действовал самостоятельно, он должен выполняться в своем собственном потоке. Java имеет превосходную встроенную поддержку потоков и позволяет использовать их мгновенно. Чтобы использовать потоки, класс Download реализует интерфейс Runnable, переопределяя метод run(). После того как метод download() создаст новый экземпляр класса Thread, передавая его конструктору класса Download, он вызывает метод start() потока. Вызов метода start() приводит к выполнению метода run() экземпляра интерфейса Runnable (класса Download).

Метод run()

При выполнении метода run() начинается процесс загрузки данных. Вследствие того, что этот метод имеет большой размер и такое же большое значение, детально проанализируем все его строки. Метод run() начинается следующим образом.

```
RandomAccessFile file = null;
InputStream stream = null;

try {
    // Открываем соединение с данным адресом URL.
    HttpURLConnection connection =
        (HttpURLConnection) url.openConnection();
```

Вначале метод run() определяет переменные для сетевого потока, из которого будет считываться содержимое, и файл для записи этого содержимого. Затем открывается соединение с адресом URL при помощи метода url.openConnection(). Поскольку известно, что утилита Download Manager поддерживает загрузку только по протоколу HTTP, соединение приводится к классу HttpURLConnection. Приведение соединения к классу HttpURLConnection позволяет воспользоваться преимуществами функциональных возможностей соединения HTTP (например, методом getResponseCode()). Обратите внимание на то, что при вызове метода url.openConnection() на самом деле не устанавливается соединение с сервером, на который указывает адрес URL. Он просто создает новый экземпляр класса URLConnection, связанный с адресом URL, который позже будет использоваться для соединения с сервером.

После того как будет создано соединение класса HttpURLConnection, при помощи вызова метода connection.setRequestProperty() определяется свойство запроса на соединение, как показано ниже.

```
// Определяем, какую часть файла нужно загрузить.
connection.setRequestProperty("Range",
    "bytes=" + downloaded + "-");
```

Если определить свойства запроса, то серверу, с которого будет производиться загрузка, можно будет посылать некоторую дополнительную информацию о запросе. В данном случае определяется свойство "Range". Это очень важное свойство, поскольку оно определяет диапазон в байтах, которые будут запрошены для загрузки с сервера. Обычно загружаются сразу все байты файла. Но если процесс загрузки будет прерван или приостановлен, получить нужно будет только оставшиеся байты. Определение свойства "Range" является основой для работы утилиты Download Manager.

Свойство "Range" определяется следующим образом.

начальный_байт - конечный_байт

Например, "0 - 12345". Однако конечный байт диапазона указывать необязательно. Если конечный байт не будет указан, диапазон завершится в конце файла. Метод run() никогда не определяет конечный байт, поскольку процессы загрузки должны выполняться до тех пор, пока не будет загружен весь диапазон, если только процесс не будет прерван или приостановлен.

Ниже показано несколько строк кода.

```
// Устанавливаем соединение с сервером.
connection.connect();
// Убеждаемся в том, что код ответа находится в диапазоне 200.
if (connection.getResponseCode() / 100 != 2) {
    error();
}

// Проверяем, имеет ли содержимое допустимую длину.
int contentLength = connection.getContentLength();
if (contentLength < 1) {
    error();
}
```

Метод connection.connect() вызывается для того, чтобы установить соединение с сервером, с которого будет производиться загрузка данных. Затем осуществляется проверка кода ответа, возвращаемого сервером. Протокол HTTP имеет список кодов ответов, которые показывают ответ сервера на запрос. Коды ответа протокола HTTP организованы в числовые диапазоны, кратные 100, а диапазон 200 указывает на успешный ответ. Чтобы проверить, находится ли код ответа в диапазоне 200, вызывается метод connection.getResponseCode() и возвращенный им результат делится на 100. Если результат этого деления будет равен 2, соединение считается успешным.

После этого метод run() получает длину содержимого вызовом метода connection.getContentLength(). Длина содержимого представляет количество байтов в требуемом файле. Если длина содержимого меньше 1, вызывается метод error(). Метод error() изменяет состояние процесса загрузки на ERROR, а затем вызывает метод stateChanged(). Мы еще вернемся к методу stateChanged().

После того как длина содержимого будет получена, следующий код проверяет, присвоено ли это значение переменной size.

```
/* Задаем размер для данного процесса загрузки,
    если он еще не был задан. */
if (size == -1) {
    size = contentLength;
    stateChanged();
}
```

Как видите, вместо того чтобы безусловно присваивать длину содержимого переменной size, она присваивается только в том случае, если переменная еще не имеет значения. Дело в том, что длина содержимого отражает количество байтов, которые будет посылать сервер. Если не будет указано ничего, кроме нулевого начального диапазона, длина содержимого будет представлять только часть размера файла. Переменной size необходимо присвоить полный размер загружаемого файла.

В следующих строках кода создается новый файл класса RandomAccessFile, для чего используется часть имени файла из адреса URL для загрузки, которая была получена с помощью метода getFileName().

```
// Открываем файл и ищем конец файла.
file = new RandomAccessFile(getFileName(url), "rw");
file.seek(downloaded);
```

Объект класса RandomAccessFile открывается в режиме "rw", который определяет, что файл можно считывать и записывать. После того как файл будет открыт, метод run() ищет конец файла с помощью метода file.seek(), присваивая значение переменной downloaded. В результате файл будет установлен на количество загруженных байтов, т.е. на конец файла. Позиционирование файла в конец необходимо на случай возобновления загрузки. При возобновлении загрузки в файл будут добавляться только новые байты, которые не будут перезаписывать ранее загруженные байты. После того как выходной файл будет подготовлен, метод connection.getInputStream() получает метку сетевого потока для открытого соединения с сервером, как показано ниже.

```
stream = connection.getInputStream();
```

Само действие начинается в цикле while.

```
while (status == DOWNLOADING) {
    /* Задаем размер буфера так, чтобы загрузить
       оставшуюся часть файла. */
    byte buffer[];
    if (size - downloaded > MAX_BUFFER_SIZE) {
        buffer = new byte[MAX_BUFFER_SIZE];
    } else {
        buffer = new byte[size - downloaded];
    }

    // Производим чтение из сервера в буфер.
    int read = stream.read(buffer);
    if (read == -1)
        break; // Записываем содержимое буфера в файл.
    file.write(buffer, 0, read);
    downloaded += read;
    stateChanged();
}
```

Цикл выполняется до тех пор, пока значением переменной status не станет DOWNLOADING. Внутри цикла создается буферный массив byte для хранения загружаемых байтов. Размер буфера определяется в соответствии с тем, какое количество данных осталось загрузить. Если загрузить осталось больше чем MAX_BUFFER_SIZE, для определения размера буфера используется значение MAX_BUFFER_SIZE. В противном случае размер буфера определяется в точности по количеству байтов, которые осталось загрузить. После того как размер буфера будет соответствующим образом определен, вызовом метода stream.read() начинается процесс загрузки. Этот метод считывает байты с сервера и переносит их в буфер, возвращая подсчет количества прочитанных байтов. Если количество прочитанных байтов равно -1, это означает, что загрузка завершена и работа цикла заканчивается. В противном случае загрузка будет продолжаться, и считываемые байты будут записываться на диск с помощью метода file.write(). Затем переменная downloaded обновляется для отображения количества байтов, загруженных на данный момент. Наконец, внутри цикла вызывается метод stateChanged(). Этот метод мы рассмотрим чуть позже.

После завершения работы цикла следующий код проверяет, по какой причине была завершена работа цикла.

```
/* Определяем состояние как завершенное, если была
   достигнута эта точка, поскольку в ней загрузка завершена. */
if (status == DOWNLOADING) {
    status = COMPLETE;
    stateChanged();
}
```

Если состоянием процесса загрузки остается DOWNLOADING, это означает, что работа цикла была завершена вследствие завершения процесса загрузки. В противном случае работа цикла была завершена по причине изменения состояния процесса загрузки.

Метод run() завершается блоками catch и finally, показанными ниже.

```
} catch (Exception e) {
    error();
} finally {

    // Закрываем файл.
    if (file != null) {
        try {
            file.close();
        } catch (Exception e) {}
    }

    // Закрываем соединение с сервером.
    if (stream != null) {
        try {
            stream.close();
        } catch (Exception e) {}
    }
}
```

Если в процессе загрузки происходит исключение, блок catch перехватит его и вызовет метод error(). Блок finally гарантирует, что если соединения file и stream были открыты, они будут закрыты вне зависимости от того, произошло исключение или нет. В качестве упражнения вы могли бы попробовать изменить данный код так, чтобы использовать для управления этими ресурсами новый оператор try-*с-ресурсами*.

Метод stateChanged()

Чтобы утилита Download Manager могла отображать самую последнюю информацию о каждом управляемом ею процессе загрузки, она должна знать обо всех изменениях информации о процессе загрузки. Для этого используется программный шаблонный проект Observer. Этот шаблон аналогичен почтовой рассылке, при которой каждый пользователь регистрируется на получение уведомлений. При появлении нового уведомления каждый человек, указанный в списке, получает сообщение с уведомлением. В случае шаблона Observer используется специальный класс, при помощи которого классы наблюдения могут зарегистрироваться на получение уведомлений об изменениях.

Класс Download использует шаблон Observer, расширяя встроенный в Java служебный класс Observable. Расширение класса Observable позволяет классам, реализующим интерфейс Java Observer, регистрироваться с помощью класса Download на получение уведомлений об изменениях. Каждый раз, когда классу Download необходимо уведомить своих зарегистрированных наблюдателей (Observer) об изменении, вызывается метод stateChanged(). Этот метод сначала вызывает метод setChanged() класса Observable, чтобы отметить

класс как такой, что был изменен. Затем метод `stateChanged()` вызывает метод `notifyObservers()` класса `Observable`, который распространяет уведомление об изменении зарегистрированным наблюдателям.

Методы действия и средства доступа

Класс `Download` содержит многочисленные методы действия и средства доступа для управления процессом загрузки и получения из него данных. Методы `pause()`, `resume()` и `cancel()`, соответственно, приостанавливают, возобновляют или отменяют процесс загрузки. Точно так же метод `error()` отмечает процесс загрузки как содержащий ошибку. Методы доступа `getURL()`, `getSize()`, `getProgress()` и `getStatus()` возвращают их текущие соответствующие значения.

Класс `ProgressRenderer`

Класс `ProgressRenderer` представляет собой небольшой служебный класс, который используется для визуализации текущего процесса загрузки, указанного в экземпляре класса `JTable` "Downloads" графического интерфейса. Обычно экземпляр класса `JTable` визуализирует данные в каждой ячейке в виде текста. Однако нередко приходится визуализировать данные в ячейке в другом виде, отличном от текстового. В случае с утилитой Download Manager нам нужно визуализировать каждую ячейку столбца Progress таблицы в виде индикатора хода работ. Это можно сделать с помощью класса `ProgressRenderer`, представленного ниже. Обратите внимание на то, что он расширяет класс `JProgressBar` и реализует интерфейс `TableCellRenderer`.

```
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;
// Этот класс визуализирует JProgressBar в ячейке таблицы.
class ProgressRenderer extends JProgressBar implements TableCellRenderer
{
    // Конструктор для ProgressRenderer.
   public ProgressRenderer(int min, int max) {
      super(min, max);
   }

   /* Возвращает JProgressBar в качестве визуализатора
      для данной ячейки таблицы. */
   public Component getTableCellRendererComponent(
               JTable table, Object value, boolean isSelected,
               boolean hasFocus, int row, int column)
   {
      // Определяет процент выполнения JProgressBar.
      setValue((int) ((Float) value).floatValue());
      return this;
   }
}
```

Класс `ProgressRenderer` использует преимущество того факта, что класс `JTable` из коллекции библиотеки Swing имеет систему визуализации, которая может принимать "подключаемые модули" для визуализации ячеек таблицы. Для использования этой системы визуализации нужно, во-первых, чтобы класс `ProgressRenderer` реализовал интерфейс `TableCellRenderer` из коллекции библиотеки Swing. Во-вторых, экземпляр `ProgressRenderer` необходимо зарегистрировать с экземпля-

ром класса JTable; так можно сообщить экземпляру класса JTable о том, какие ячейки необходимо визуализировать с помощью "подключаемого модуля".

Для реализации интерфейса TableCellRenderer необходимо, чтобы класс переопределял метод getTableCellRendererComponent().

Метод getTableCellRendererComponent() вызывается каждый раз, когда экземпляр класса JTable приступает к визуализации ячейки, для которой зарегистрирован этот класс. Этот метод передает несколько переменных, хотя в данном случае используется только переменная value. Эта переменная хранит данные для визуализируемой ячейки и передается методу setValue() класса JProgressBar. В результате выполнения метода getTableCellRendererComponent() возвращается ссылка на его класс. Это возможно благодаря тому, что класс ProgressRenderer является подклассом класса JProgressBar, который представляет собой наследника класса Component библиотеки AWT.

Класс DownloadsTableModel

Класс DownloadsTableModel содержит список процессов загрузки утилиты Download Manager и является резервным источником данных для экземпляра класса JTable "Downloads" графического интерфейса.

Ниже представлен класс DownloadsTableModel. Обратите внимание на то, что он расширяет класс AbstractTableModel и реализует интерфейс Observer.

```
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

// Этот класс управляет данными таблицы загрузки.
class DownloadsTableModel extends AbstractTableModel
implements Observer
{

    // Имена столбцов таблицы.
    private static final String[] columnNames = {"URL", "Size",
                                                 "Progress", "Status"};

    // Классы для значений каждого столбца.
    private static final Class[] columnClasses = {String.class,
                        String.class, JProgressBar.class, String.class};

    // Табличный список процессов загрузки.
    private ArrayList<Download> downloadList = new ArrayList<Download>();

    // Добавление нового процесса загрузки в таблицу.
    public void addDownload(Download download) {
        // Регистрация на получение уведомления об изменениях
        // в процессе загрузки.
        download.addObserver(this);

        downloadList.add(download);

        // Создание для таблицы уведомления о вставке строки.
        fireTableRowsInserted(getRowCount() - 1, getRowCount() - 1);
    }

    // Получение процесса загрузки для определенной строки.
    public Download getDownload(int row) {
        return downloadList.get(row);
    }
```

```
    // Удаление процесса загрузки из списка.
    public void clearDownload(int row) {
        downloadList.remove(row);

        // Создание для таблицы уведомления об удалении строки.
        fireTableRowsDeleted(row, row);
    }

    // Получение подсчитанного количества столбцов таблицы.
    public int getColumnCount() {
        return columnNames.length;
    }

    // Получение имени столбца.
    public String getColumnName(int col) {
        return columnNames[col];
    }

    // Получение класса столбца.
    public Class getColumnClass(int col) {
        return columnClasses[col];
    }

    // Получение подсчитанного количества строк таблицы.
    public int getRowCount() {
        return downloadList.size();
    }
    // Получение значения для определенной комбинации строки и столбца.
    public Object getValueAt(int row, int col) {
        Download download = downloadList.get(row);
        switch (col) {
            case 0: // адрес URL
                return download.getUrl();

            case 1: // Размер
                int size = download.getSize();
                return (size == -1) ? "" : Integer.toString(size);
            case 2: // Ход выполнения
                return new Float(download.getProgress());
            case 3: // Состояние
                return Download.STATUSES[download.getStatus()];
        }
        return "";
    }

    /* Вызываем обновление, если процесс загрузки сообщает
       своим наблюдателям о каких-либо изменениях */
    public void update(Observable o, Object arg) {
        int index = downloadList.indexOf(o);

        // Создание для таблицы уведомления об обновлении строки.
        fireTableRowsUpdated(index, index);
    }
}
```

Класс `DownloadsTableModel`, по сути, является служебным классом, используемым экземпляром класса `JTable` "Downloads" для управления данными в таблице. После того как экземпляр класса `JTable` инициализирован, он передается экземпляру класса `DownloadsTableModel`. Затем экземпляр класса `JTable` вызывает несколько методов экземпляра класса `DownloadsTableModel` для заполнения таблицы. Метод `getColumnCount()` вызывается для получения количества столб-

цов в таблице. Подобно ему, метод `getRowCount()` используется для получения количества строк в таблице. Метод `getColumnName()` возвращает имя столбца на основании его идентификатора. Метод `getDownload()` принимает идентификатор строки и возвращает связанный объект класса `Download` из списка. Об остальных методах класса `DownloadsTableModel`, которые являются более сложными, поговорим подробно в следующих разделах.

Метод `addDownload()`

Показанный ниже метод `addDownload()` добавляет новый объект класса `Download` в список управляемых процессов загрузки и, следовательно, строку в таблицу.

```
// Добавление нового процесса загрузки в таблицу.
public void addDownload(Download download) {
    // Регистрация на получение уведомления об изменениях
    // в процессе загрузки.
    download.addObserver(this);

    downloadList.add(download);

    // Создание для таблицы уведомления о вставке строки.
    fireTableRowsInserted(getRowCount() - 1, getRowCount() - 1);
}
```

Этот метод сначала регистрируется на получение уведомлений об изменениях с новым классом `Download` в качестве наблюдателя. Затем объект класса `Download` добавляется во внутренний список управляемых процессов загрузки. В конце метода создается извещение о событии вставки строки в таблицу, чтобы оповестить таблицу о добавлении новой строки.

Метод `clearDownload()`

Этот метод удаляет объект класса `Download` из списка управляемых процессов загрузки.

```
// Удаление процесса загрузки из списка.
public void clearDownload(int row) {
    downloadList.remove(row);

    // Создание для таблицы уведомления об удалении строки.
    fireTableRowsDeleted(row, row);
}
```

После удаления объекта класса `Download` из внутреннего списка создается извещение о событии удаления строки таблицы, чтобы оповестить таблицу об удалении строки.

Метод `getColumnClass()`

Представленный ниже метод `getColumnClass()` возвращает тип класса для данных, отображаемых в определенном столбце.

```
// Получение класса столбца.
public Class<?> getColumnClass(int col) {
    return columnClasses[col];
}
```

Все столбцы отображаются в текстовом виде (т.е. как объекты класса `String`), за исключением столбца Progress, который отображается в виде индикатора хода работ (который является объектом класса `JProgressBar`).

Метод `getValueAt()`

Этот метод вызывается для получения текущего значения, которое необходимо отобразить в каждой ячейке таблицы.

```
// Получение значения для определенной комбинации строки и столбца.
public Object getValueAt(int row, int col) {
    Download download = downloadList.get(row);
    switch (col) {
        case 0: // адрес URL
            return download.getUrl();
        case 1: // Размер
            int size = download.getSize();
            return (size == -1) ? "" : Integer.toString(size);
        case 2: // Ход выполнения
            return new Float(download.getProgress());
        case 3: // Состояние
            return Download.STATUSES[download.getStatus()];
    }
    return "";
}
```

Этот метод сначала ищет объект класса `Download`, соответствующий указанной строке. Затем указанный столбец используется для определения того, какое из значений свойств объекта класса `Download` необходимо вернуть.

Метод `update()`

Метод `update()`, показанный ниже, заполняет контракт интерфейса `Observer`, позволяя классу `DownloadsTableModel` получать уведомления от объектов класса `Download` в случае их изменения.

```
/* Вызываем обновление, если процесс загрузки сообщает
   своим наблюдателям о каких-либо изменениях */
public void update(Observable o, Object arg) {
    int index = downloadList.indexOf(o);

    // Создание для таблицы уведомления об обновлении строки.
    fireTableRowsUpdated(index, index);
}
```

Этот метод передает ссылку на измененный объект класса `Download` в виде объекта класса `Observable`. Затем в списке процессов загрузки осуществляется поиск индекса этого процесса загрузки. Впоследствии этот индекс будет использоваться для создания извещений о событии обновления строки таблицы, которое известит таблицу об обновлении данной строки. После этого таблица повторно визуализирует строку с данным индексом, отображая ее новые значения.

Класс `DownloadManager`

Теперь, после того как вы детально ознакомились с вспомогательными классами утилиты Download Manager, можно переходить к рассмотрению класса

DownloadManager, который отвечает за создание и запуск графического интерфейса утилиты Download Manager. Этот класс объявляет метод main(), поэтому при выполнении он будет вызываться первым. Метод main() реализует новый экземпляр класса DownloadManager, а затем вызывает его метод show(), который и приводит к его отображению.

Класс DownloadManager показан ниже. Обратите внимание на то, что он расширяет класс JFrame и реализует интерфейс Observer. В следующих разделах проанализируем его подробнее.

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

// Класс утилиты Download Manager.
public class DownloadManager extends JFrame
    implements Observer
{

    // Добавление текстового поля процесса загрузки.
    private JTextField addTextField;

    // Модель данных таблицы процесса загрузки.
    private DownloadsTableModel tableModel;

    // Таблица с перечислением процессов загрузки.
    private JTable table;\

    // Кнопки для управления выделенным процессом загрузки.
    private JButton pauseButton, resumeButton;
    private JButton cancelButton, clearButton;
    // Выделенный на данный момент процесс загрузки.
    private Download selectedDownload;
    // Флаг, обозначающий, очищается ли выбор таблицы.
    private boolean clearing;
    // Конструктор для Download Manager.
    public DownloadManager()
    {
        // Определение заголовка приложения.
        setTitle("Download Manager");

        // Определение размеров окна.
        setSize(640, 480);

        // Обработка событий закрытия окна.
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                actionExit();
            }
        });

        // Настройка меню файлов.
        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("File");
        fileMenu.setMnemonic(KeyEvent.VK_F);
        JMenuItem fileExitMenuItem = new JMenuItem("Exit",
                                                   KeyEvent.VK_X);
        fileExitMenuItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
            actionExit();
```

```
        }
    });
    fileMenu.add(fileExitMenuItem);
    menuBar.add(fileMenu);
    setJMenuBar(menuBar);

    // Настройка панели добавления.
    JPanel addPanel = new JPanel();
    addTextField = new JTextField(30);
    addPanel.add(addTextField);
    JButton addButton = new JButton("Add Download");
    addButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            actionAdd();
        }
    });
    addPanel.add(addButton);

    // Настройка таблицы Downloads.
    tableModel = new DownloadsTableModel();
    table = new JTable(tableModel);
    table.getSelectionModel().addListSelectionListener(new
                                ListSelectionListener() {
        public void valueChanged(ListSelectionEvent e) {
            tableSelectionChanged();
        }
    });

    // Разрешает выбрать в одно и то же время только одну строку.
    table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    // Настройка ProgressBar в качестве визуализатора для столбца
    // хода выполнения.
    ProgressRenderer renderer = new ProgressRenderer(0, 100);
    renderer.setStringPainted(true); // показывает текст хода выполнения
    table.setDefaultRenderer(JProgressBar.class, renderer);

    // Определение высоты строки таблицы, достаточной для того,
    // чтобы уместить JProgressBar.
    table.setRowHeight(
        (int) renderer.getPreferredSize().getHeight());

        // Настройка панели процессов загрузки.
        JPanel downloadsPanel = new JPanel();
        downloadsPanel.setBorder(
            BorderFactory.createTitledBorder("Downloads"));
            downloadsPanel.setLayout(new BorderLayout());
            downloadsPanel.add(new JScrollPane(table),
                BorderLayout.CENTER);

            // Настройка панели кнопок.
            JPanel buttonsPanel = new JPanel();
            pauseButton = new JButton("Pause");
            pauseButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    actionPause();
                }
            });
            pauseButton.setEnabled(false);
            buttonsPanel.add(pauseButton);
            resumeButton = new JButton("Resume");
            resumeButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
```

```
                actionResume();
            }
        });
        resumeButton.setEnabled(false);
        buttonsPanel.add(resumeButton);
        cancelButton = new JButton("Cancel");
        cancelButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                actionCancel();
            }
        });
        cancelButton.setEnabled(false);
        buttonsPanel.add(cancelButton);
        clearButton = new JButton("Clear");
        clearButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                actionClear();
            }
        });
        clearButton.setEnabled(false);
        buttonsPanel.add(clearButton);

        // Добавление панелей для отображения.
        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(addPanel, BorderLayout.NORTH);
        getContentPane().add(downloadsPanel, BorderLayout.CENTER);
        getContentPane().add(buttonsPanel, BorderLayout.SOUTH);
    }

    // Выход из программы.
    private void actionExit() {
        System.exit(0);
    }

    // Добавление нового процесса загрузки.
    private void actionAdd() {
        URL verifiedUrl = verifyUrl(addTextField.getText());
        if (verifiedUrl != null) {
            tableModel.addDownload(new Download(verifiedUrl));
            addTextField.setText(""); // сброс добавления
                                      // текстового поля
        } else {
            JOptionPane.showMessageDialog(this,
                                "Invalid Download URL", "Error",
                                JOptionPane.ERROR_MESSAGE);
        }
    }

    // Проверка адреса URL для загрузки.
    private URL verifyUrl(String url) {
        // Разрешаем только те адреса URL, которые включают HTTP.
        if (!url.toLowerCase().startsWith("http://"))
            return null;

        // Проверка формата адреса URL.
        URL verifiedUrl = null;
        try {
            verifiedUrl = new URL(url);
        } catch (Exception e) {
            return null;
        }

        // Проверяем, содержит ли адрес URL файл.
```

```
        if (verifiedUrl.getFile().length() < 2)
            return null;
        return verifiedUrl;
    }

    // Вызывается в том случае, если изменяется выбор строки таблицы.
    private void tableSelectionChanged() {
        /* Отказ от регистрации на получение уведомлений
           от последнего выбранного процесса загрузки. */
        if (selectedDownload != null)
            selectedDownload.deleteObserver(DownloadManager.this);

        /* Если не происходит очистка процесса загрузки, задаем
           выбранный процесс загрузки и регистрируемся на получение
           от него уведомлений. */
        if (!clearing && table.getSelectedRow() > -1) {
            selectedDownload =
                tableModel.getDownload(table.getSelectedRow());

            selectedDownload.addObserver(DownloadManager.this);
            updateButtons();
        }
    }

    // Приостановка выбранного процесса загрузки.
    private void actionPause() {
        selectedDownload.pause();
        updateButtons();
    }

    // Возобновление выбранного процесса загрузки.
    private void actionResume() {
        selectedDownload.resume();
        updateButtons();
    }

    // Отмена выбранного процесса загрузки.
    private void actionCancel() {
        selectedDownload.cancel();
        updateButtons();
    }

    // Очистка выбранного процесса загрузки.
    private void actionClear() {
        clearing = true;
        tableModel.clearDownload(table.getSelectedRow());
        clearing = false;
        selectedDownload = null;
        updateButtons();
    }

    /* Обновление состояния каждой кнопки на основании состояния
       выбранного на данный момент процесса загрузки. */
    private void updateButtons() {
        if (selectedDownload != null) {
            int status = selectedDownload.getStatus();
            switch (status) {
                case Download.DOWNLOADING:
                    pauseButton.setEnabled(true);
                    resumeButton.setEnabled(false);
                    cancelButton.setEnabled(true);
                    clearButton.setEnabled(false);
                    break;
```

```
                case Download.PAUSED:
                    pauseButton.setEnabled(false);
                    resumeButton.setEnabled(true);
                    cancelButton.setEnabled(true);
                    clearButton.setEnabled(false);
                    break;
                case Download.ERROR:
                    pauseButton.setEnabled(false);
                    resumeButton.setEnabled(true);
                    cancelButton.setEnabled(false);
                    clearButton.setEnabled(true);
                    break;
                default: // COMPLETE или CANCELLED
                    pauseButton.setEnabled(false);

                    resumeButton.setEnabled(false);
                    cancelButton.setEnabled(false);
                    clearButton.setEnabled(true);
            }
        } else {
            // В таблице не выбрано ни одного процесса загрузки.
            pauseButton.setEnabled(false);
            resumeButton.setEnabled(false);
            cancelButton.setEnabled(false);
            clearButton.setEnabled(false);
        }
    }

    /* Вызывается обновление, когда объект Download уведомляет
       своих наблюдателей о каких-либо изменениях. */
    public void update(Observable o, Object arg) {
        // Обновление кнопок в случае изменения выбранного
        // процесса загрузки.
        if (selectedDownload != null && selectedDownload.equals(o))
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    updateButtons();
                }
            });
    }

    // Запуск утилиты Download Manager.
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                DownloadManager manager = new DownloadManager();
                manager.setVisible(true);
            }
        });
    }
}
```

Переменные класса `DownloadManager`

Класс `DownloadManager` начинается с объявления нескольких переменных экземпляра, большинство из которых хранит ссылки на элементы управления графического интерфейса. Переменная `selectedDownload` хранит ссылку на объект класса `Download`, представляющую выбранную строку в таблице. Переменная экземпляра `clearing` является флагом булева типа, который отмечает, удаляется ли процесс загрузки из таблицы Downloads.

Конструктор класса DownloadManager

В процессе реализации класса DownloadManager все элементы управления графического интерфейса инициализируются внутри его конструктора. Хотя конструктор и содержит большую часть кода, на самом деле он прост. Сейчас вы в этом убедитесь.

Во-первых, с помощью метода setTitle() определяется заголовок окна. При вызове метода setSize() задается высота и ширина окна в пикселях. После этого добавляется слушатель окна вызовом метода addWindowListener() с передачей ему объекта класса WindowAdapter, в котором переопределен обработчик событий windowClosing(). Этот обработчик вызывает метод actionExit() во время закрытия окна приложения. В окно приложения добавляется строка меню, содержащая меню File (Файл). Затем устанавливается панель Add (Добавить), которая содержит поле Add Text (Добавить текст) и кнопку. Для кнопки Add Download (Добавить процесс загрузки) добавляется слушатель ActionListener, чтобы можно было вызывать метод actionAdd() при каждом щелчке на кнопке.

После этого создается таблица процессов загрузки. В эту таблицу добавляется слушатель ListSelectionListener, благодаря которому при выборе строки в таблице будет вызываться метод tableSelectionChanged(). Режим выбора таблицы также изменяется на ListSelectionModel.SINGLE_SELECTION, поэтому в таблице можно выделить за один раз только одну строку. Выделение только одной строки упрощает логику, определяющую, какие кнопки необходимо сделать активными в интерфейсе при выделении строки в таблице процесса загрузки. Затем создается экземпляр класса ProgressRenderer и регистрируется с таблицей для обработки столбца "Progress". Высота строки таблицы обновляется до высоты ProgressRenderer при помощи метода table.setRowHeight(). После того как таблица готова, с помощью класса JScrollPane в нее добавляются полосы прокрутки, после чего она помещается на панель.

В завершение создается панель кнопок. На ней имеются кнопки Pause (Приостановить), Resume (Возобновить), Cancel (Отменить) и Clear (Очистить). Каждая кнопка добавляет слушатель ActionListener, который при щелчке на данной кнопке вызывает соответствующий метод действия. После создания панели кнопок все созданные панели добавляются в окно.

Метод verifyUrl()

Метод verifyUrl() вызывается методом actionAdd() при добавлении процесса загрузки в утилиту Download Manager. Этот метод показан ниже.

```
// Проверка адреса URL для загрузки.
private URL verifyUrl(String url) {
    // Разрешаем только те адреса URL, которые включают HTTP.
    if (!url.toLowerCase().startsWith("http://"))
        return null;

    // Проверка формата адреса URL.
    URL verifiedUrl = null;
    try {
        verifiedUrl = new URL(url);
    } catch (Exception e) {
        return null;
    }

    // Проверяем, содержит ли адрес URL файл.
    if (verifiedUrl.getFile().length() < 2)
        return null;
```

```
    return verifiedUrl;
}
```

Этот метод сначала проверяет, соответствует ли введенный адрес URL протоколу HTTP, поскольку из всех протоколов поддерживается только он. После проверки адрес URL используется для создания нового экземпляра класса `URL`. Если адрес URL составлен неправильно, конструктор класса `URL` передает исключение. В завершение этот метод проверяет, содержит ли адрес URL файл.

Метод `tableSelectionChanged()`

Метод `tableSelectionChanged()`, показанный ниже, вызывается каждый раз при выделении строки в таблице процессов загрузки.

```
// Вызывается в том случае, если изменяется выбор строки таблицы.
private void tableSelectionChanged() {
    /* Отказ от регистрации на получение уведомлений
       от последнего выбранного процесса загрузки. */
    if (selectedDownload != null)
        selectedDownload.deleteObserver(DownloadManager.this);

    /* Если не происходит очистка процесса загрузки, задаем
       выбранный процесс загрузки и регистрируемся на получение
       от него уведомлений. */
    if (!clearing && table.getSelectedRow() > -1) {
        selectedDownload =
            tableModel.getDownload(table.getSelectedRow());
        selectedDownload.addObserver(DownloadManager.this);
        updateButtons();
    }
}
```

Работа метода начинается с того, что он проверяет, выделена ли в данный момент какая-либо строка. Для этого он проверяет, не имеет ли переменная `selectedDownload` значение `null`. Если переменная `selectedDownload` не содержит значение `null`, то экземпляр класса `DownloadManager` больше не будет получать уведомления в качестве наблюдателя. Затем проверяется флаг `clearing`. Если таблица не является пустой и флаг `clearing` содержит значение `false`, то переменной `selectedDownload` присваивается объект класса `Download`, соответствующий выделенной строке. После этого экземпляр класса `DownloadManager` регистрируется как наблюдатель с новым выбранным объектом класса `Download`. В конце вызывается метод `updateButtons()` для обновления состояний кнопок на основании выбранного состояния объекта класса `Download`.

Метод `updateButtons()`

Метод `updateButtons()` обновляет состояние всех кнопок на панели кнопок на основе состояния выделенного процесса загрузки. Метод `updateButtons()` показан ниже.

```
/* Обновление состояния каждой кнопки на основании состояния
   выбранного на данный момент процесса загрузки. */
private void updateButtons() {
    if (selectedDownload != null) {
        int status = selectedDownload.getStatus();
        switch (status) {
            case Download.DOWNLOADING:
                pauseButton.setEnabled(true);
```

```
                resumeButton.setEnabled(false);
                cancelButton.setEnabled(true);
                clearButton.setEnabled(false);
                break;
            case Download.PAUSED:
                pauseButton.setEnabled(false);
                resumeButton.setEnabled(true);
                cancelButton.setEnabled(true);
                clearButton.setEnabled(false);
                break;
            case Download.ERROR:
                pauseButton.setEnabled(false);
                resumeButton.setEnabled(true);
                cancelButton.setEnabled(false);
                clearButton.setEnabled(true);
                break;
            default: // COMPLETE или CANCELLED
                pauseButton.setEnabled(false);
                resumeButton.setEnabled(false);
                cancelButton.setEnabled(false);
                clearButton.setEnabled(true);
        }
    } else {
        // В таблице не выбрано ни одного процесса загрузки.
        pauseButton.setEnabled(false);
        resumeButton.setEnabled(false);
        cancelButton.setEnabled(false);
        clearButton.setEnabled(false);
    }
}
```

Если в таблице не выбрано ни одного процесса загрузки, все кнопки становятся неактивными и окрашиваются серым цветом. Если процесс загрузки будет выделен, то состояние каждой кнопки будет обновлено на основе состояния объекта класса `Download` — `DOWNLOADING`, `PAUSED`, `ERROR`, `COMPLETE` или `CANCELLED`.

Обработка событий действий

Каждый элемент управления графического интерфейса утилиты Download Manager регистрирует слушатель `ActionListener`, который вызывает соответствующий метод действия. Слушатели `ActionListener` запускаются каждый раз, когда происходит событие действия в элементе управления графического интерфейса. Например, если пользователь щелкает на кнопке, происходит событие класса `ActionEvent`, о котором уведомляется каждый зарегистрированный слушатель кнопки `ActionListener`. Вы могли заметить сходство в работе слушателей `ActionListener` и рассмотренного ранее шаблона Observer. Это объясняется тем, что они представляют собой один и тот же шаблон, но имеют разные схемы именования.

Компиляция и запуск утилиты Download Manager

Компиляция утилиты Download Manager выполняется следующим образом.

```
javac DownloadManager.java DownloadsTableModel.java ProgressRenderer.
javaDownload.java
```

Запуск утилиты Download Manager производится так.

```
javaw DownloadManager
```

Использовать утилиту Download Manager несложно. Для начала в текстовом поле в верхней части экрана нужно ввести адрес URL файла, который требуется загрузить.

Например, чтобы загрузить файл `0072229713_code.zip`, хранящийся на сайте `www.osborne.com`, нужно ввести следующее.

```
http://www.mhprofessional.com/downloads/products/0072229713/
0072229713_code.zip
```

Это файл, в котором содержится код для моей книги *The Art of Java*, которую я написал в соавторстве с Джеймсом Холмсом (James Holmes).

После того как процесс загрузки будет добавлен в утилиту Download Manager, вы сможете управлять им, выделяя его в таблице. После того как процесс будет выделен, вы можете приостановить его, отменить, возобновить и очистить. На рис. 34.2 показана утилита Download Manager в действии.

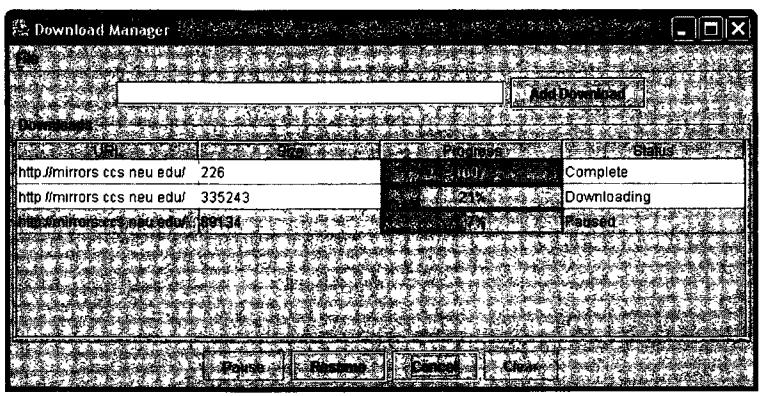


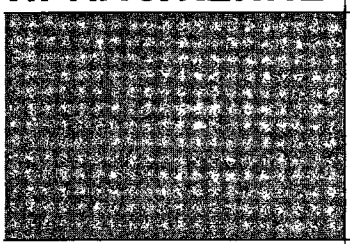
Рис. 34.2. Утилита Download Manager в действии

Расширение утилиты Download Manager

Утилита Download Manager представляет собой полнофункциональное средство, позволяющее приостанавливать и возобновлять процессы загрузки, а также загружать несколько файлов одновременно. Тем не менее вы можете расширить ее возможности и сделать еще лучше. Смотрите, что еще можно сделать: добавить поддержку прокси-серверов, протоколов FTP и HTTPS, а также поддержку технологии перетаскивания. Интересной особенностью является планирование, благодаря которому сможете запланировать процесс загрузки на определенное время (например, на час ночи), когда можно использовать большое количество свободных системных ресурсов.

Учтите, что продемонстрированные в этой главе методы не ограничены загрузкой файлов в обычном смысле. Этому коду можно найти еще множество применений. Например, многие программы, распространяемые через Интернет, поступают в двух частях. Первая часть имеет небольшой размер и представляет собой компактное приложение, которое можно загрузить очень быстро. Это приложение содержит мини-диспетчер загрузки для загрузки второй части, которая обычно имеет гораздо большие размеры. Такой подход удобно использовать для больших приложений, при загрузке которых возрастает вероятность прерывания. Возможно, вы решите адаптировать утилиту Download Manager для этих целей.

ПРИЛОЖЕНИЕ



Использование комментариев документации

Как было сказано в части I, язык Java поддерживает три типа комментариев. Первые два – это комментарии `//` и `/* */`. Третий тип называется *комментарием документации*. Такой комментарий начинается с последовательности символов `/**` и заканчивается последовательностью `*/`. Комментарии документации позволяют добавлять в программу информацию о ней самой. Позже с помощью утилиты `javadoc` (входящей в состав JDK) эту информацию можно будет извлекать и помещать в файл HTML. Комментарии документации облегчают процесс написания документации к разрабатываемым программам. Вы, должно быть, встречали документацию, созданную утилитой `javadoc`, поскольку именно она использована для документирования библиотеки API Java.

Дескрипторы утилиты `javadoc`

Утилита `javadoc` распознает дескрипторы, перечисленные в табл. 1.

Таблица 1. Дескрипторы утилиты `javadoc`

Дескриптор	Назначение
`@author`	Идентифицирует автора
`{@code}`	Отображает информацию "как есть", без обработки стилей HTML, используя шрифт кода
`@deprecated`	Определяет, что класс или его член является устаревшим
`{@docRoot}`	Определяет путь к корневому каталогу текущей документации
`@exception`	Идентифицирует исключение, переданное методом или конструктором
`{@inheritDoc}`	Наследует комментарий от непосредственного суперкласса
`{@link}`	Вставляет встроенную ссылку на другую тему
`{@linkplain}`	Вставляет встроенную ссылку на другую тему, при этом ссылка отображается обычным шрифтом
`{@literal}`	Отображает информацию "как есть", без обработки стилей HTML
`@param`	Документирует параметр метода
`@return`	Документирует возвращаемое значение метода
`@see`	Определяет ссылку на другую тему
`@serial`	Документирует поле, сериализируемое по умолчанию
`@serialData`	Документирует данные, записанные методами `writeObject()` и `writeExternal()`

Окончание табл. 1

Дескриптор	Назначение
`@serialField`	Документирует компонент `ObjectStreamField`
`@since`	Показывает, в каком выпуске было введено определенное изменение
`@throws`	То же, что и дескриптор `@exception`
`{@value}`	Отображает значение константы, которая должна быть статическим полем
`@version`	Определяет версию класса

Дескрипторы `javadoc`, начинающиеся со знака `@`, называются автономными (а также дескрипторами блока) и должны использоваться в своей собственной строке. Дескрипторы, начинающиеся с фигурной скобки, например `{@code}`, называются встроенными и могут применяться внутри большего описания. В комментариях документации можно использовать и другие стандартные дескрипторы HTML. Однако некоторые дескрипторы, такие как заголовки, нельзя использовать, потому что они нарушают вид файла HTML, сформированный утилитой `javadoc`.

Комментарии документации можно применять для документирования классов, интерфейсов, полей, конструкторов и методов. В каждом из случаев комментарий документации должен стоять перед документируемым элементом. Некоторые дескрипторы, такие как `@see`, `@since` и `@deprecated`, применяются для документирования любого элемента. Другие дескрипторы применимы только к соответствующим элементам. Сейчас рассмотрим каждый из этих дескрипторов.

На заметку! Комментарии могут быть также использованы для документирования пакета и подготовки краткого обзора, но эти процедуры отличаются от используемых для документирования исходного кода. Более подробная информация об этом содержится в документации на утилиту `javadoc`.

Дескриптор `$author`

Этот дескриптор документирует автора класса. Он имеет следующий синтаксис.

```
@author описание
```

Здесь *описание* обычно представляет фамилию человека, написавшего класс. При выполнении утилиты `javadoc` вам нужно будет задать параметр `-author`, чтобы включить в документацию HTML поле дескриптора `@author`.

Дескриптор `{@code}`

Дескриптор `{@code}` позволяет встраивать в комментарий текст (например, фрагмент кода). Этот текст будет отображаться шрифтом кода без последующей обработки (например, без визуализации HTML). Он имеет следующий синтаксис.

```
{@code фрагмент_кода}
```

Дескриптор `@deprecated`

Дескриптор `@deprecated` определяет устаревший программный элемент. Чтобы информировать программиста о доступных альтернативных вариантах, рекомендуется включать дескрипторы `@see` или `{@link}`. Синтаксис этого дескриптора выглядит следующим образом.

```
@deprecated описание
```

Здесь *описание* — это сообщение, описывающее исключение. Дескриптор `@deprecated` может использоваться для документирования полей, методов, конструкторов и классов.

Дескриптор {@docRoot}

Дескриптор `{@docRoot}` определяет путь к корневому каталогу текущей документации.

Дескриптор @exception

Дескриптор `@exception` описывает исключение для данного метода. Он имеет следующий синтаксис.

`@exception` *имя_исключения пояснение*

Здесь *имя_исключения* указывает полное имя исключения, а *пояснение* представляет строку, которая описывает, в каких случаях может произойти данное исключение. Дескриптор `@exception` может использоваться только для документирования методов или конструкторов.

Дескриптор {@inheritDoc}

Этот дескриптор наследует комментарий от непосредственного суперкласса.

Дескриптор {@link}

Дескриптор `{@link}` предлагает встроенную ссылку на дополнительную информацию. Он имеет следующий синтаксис.

`{@link` *пакет.класс#член текст*`}`

Здесь *пакет.класс#член* определяет имя класса или метода, на который добавляется ссылка, а *текст* представляет отображаемую строку.

Дескриптор {@linkplain}

Вставляет встроенную ссылку на другую тему. Ссылка отображается обычным шрифтом. В противном случае дескриптор аналогичен `{@link}`.

Дескриптор {@literal}

Дескриптор `{@literal}` позволяет встраивать текст в комментарий. Этот текст отображается "как есть" без последующей обработки (например, без визуализации HTML). Он имеет следующий синтаксис.

`{@literal` *описание*`}`

Здесь *описание* представляет встраиваемый текст.

Дескриптор @param

Дескриптор `@param` документирует параметр. Он имеет следующий синтаксис.

`@param` *имя_параметра пояснение*

Здесь *имя_параметра* представляет имя параметра. Назначение этого параметра определяется соответствующим пояснением. Дескриптор `@param` может использоваться только для документирования метода, конструктора или обобщенного класса или интерфейса.

Дескриптор `@return`

Дескриптор `@return` описывает возвращаемое значение метода. Он имеет следующий синтаксис.

```
@return пояснение
```

Здесь *пояснение* описывает тип и смысл значения, возвращаемого методом. Дескриптор `@return` может использоваться только для документирования метода.

Дескриптор `@see`

Дескриптор `@see` обеспечивает ссылку на дополнительную информацию. Далее показаны наиболее распространенные его формы.

```
@see привязка
@see пакет.класс#член текст
```

В первой форме *привязка* представляет ссылку на абсолютный или относительный адрес URL. Во второй форме *пакет.класс#член* определяет имя элемента, а *текст* представляет отображаемый для данного элемента текст. Текстовый параметр необязателен, и если он не используется, то отображается элемент, указанный в параметре *пакет.класс#член*. Имя члена тоже является необязательным. Таким образом, вы можете определить ссылку на пакет, класс или интерфейс в дополнение к ссылке на определенные метод или поле. Имя может быть определено полностью или частично. Однако точку, стоящую перед именем члена (если таковой существует), необходимо заменить символом #.

Дескриптор `@serial`

Дескриптор `@serial` определяет комментарий для поля, сериализируемого по умолчанию. Он имеет следующий синтаксис.

```
@serial описание
```

Здесь *описание* определяет комментарий для данного поля.

Дескриптор `@serialData`

Дескриптор `@serialData` документирует данные, записанные с помощью методов `writeObject()` и `writeExternal()`. Он имеет следующий синтаксис.

```
@serialData описание
```

Здесь *описание* определяет комментарий для этих данных.

Дескриптор `@serialField`

Для класса, реализующего интерфейс `Serializable`, дескриптор `@serialField` предлагает комментарии для компонента `ObjectStreamField`. Он имеет следующий синтаксис.

```
@serialField имя тип описание
```

Здесь *имя* представляет имя поля, *тип* — его тип, а *описание* — комментарий для данного поля.

Дескриптор @since

Дескриптор `@since` показывает, что класс или элемент был впервые представлен в определенном выпуске. Он имеет следующий синтаксис.

```
@since выпуск
```

Здесь *выпуск* представляет строку, в которой указаны выпуск или версия, начиная с которых эта возможность стала доступной.

Дескриптор @throws

Дескриптор `@throws` имеет то же назначение, что и дескриптор `@exception`.

Дескриптор {@value}

Дескриптор `{@value}` имеет две формы. Первая отображает значение следующей за ним константы, которой должно быть статическое поле. Его форма показана ниже.

```
{@value}
```

Вторая форма отображает значение определенного статического поля. В этом случае дескриптор имеет следующую форму.

```
{@value пакет.класс#поле}
```

Здесь *пакет.класс#поле* определяет имя статического поля.

Дескриптор @version

Дескриптор `@version` определяет версию класса или интерфейса. Он имеет следующий синтаксис.

```
@version информация
```

Здесь *информация* представляет строку, содержащую информацию о версии (как правило, номер версии, например 2.2). При запуске утилиты `javadoc` вам нужно будет указать параметр `-version`, чтобы включить в документацию HTML поле `@version`.

Общая форма комментариев документации

После начальной комбинации символов `/**` первая строка (или строки) становится главным описанием вашего класса, интерфейса, поля, конструктора или метода. После нее можно включать один или более различных дескрипторов `@`. Каждый дескриптор `@` должен стоять в начале новой строки или следовать за одним или несколькими символами звездочки (`*`), находящимися в начале строки. Несколько дескрипторов одного и того же типа необходимо группировать вместе. Например, если у вас имеется три дескриптора `@see`, их следует поместить друг

за другом. Встроенные дескрипторы (они начинаются с фигурной скобки) можно помещать внутри любого описания.

Ниже показан пример комментария документации для класса.

```
/**
* Этот класс рисует секторную диаграмму.
* @author Герберт Шилдт
* @version 3.2
*/
```

Вывод утилиты javadoc

Утилита javadoc в качестве входных данных получает файл с исходным кодом вашей программы Java и выводит несколько файлов HTML, содержащих документацию по этой программе. Информация о каждом классе будет содержаться в его собственном файле HTML. Утилита javadoc выводит также дерево индексов и иерархии. Могут быть созданы и другие файлы HTML.

Пример использования комментариев документации

Ниже приведена простая программа, в которой используются комментарии документации. Обратите внимание на то, что каждый комментарий стоит непосредственно перед тем элементом, который он описывает. После того как документация по классу SquareNum будет обработана утилитой javadoc, ее можно будет найти в файле SquareNum.html.

```
import java.io.*;
/**
* Этот класс демонстрирует применение комментариев документации.
* @author Герберт Шилдт (Herbert Schildt)
* @version 1.2
*/
public class SquareNum {
    /**
    * Этот метод возвращает квадрат числа.
    * Это многострочное описание. Вы можете использовать
    * столько строк, сколько будет необходимо.
    * @param num Значение, которое необходимо возвести в квадрат.
    * @return num Значение, возведенное в квадрат.
    */
    public double square(double num) {
        return num * num;
    }

    /**
    * Этот метод вводит число, полученное от пользователя.
    * @return Введенное значение в виде double.
    * @exception В случае ошибки ввода передается исключение IOException.
    * @see IOException
    */
    public double getNumber() throws IOException {
        // Создает BufferedReader с помощью 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 square().
    * @param args Не используется.
    * @exception В случае ошибки ввода передается исключение IOException.
    * @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("Значение в квадрате равно " + val);
    }
}
```

Предметный указатель

К

Л

М

H

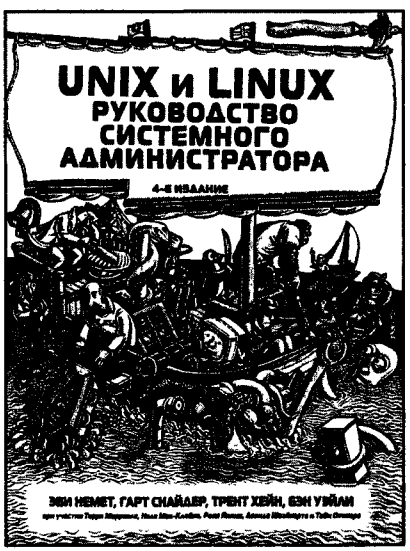
O

П

Р

UNIX И LINUX
РУКОВОДСТВО СИСТЕМНОГО АДМИНИСТРАТОРА
4-Е ИЗДАНИЕ

Эви Немет
Гарт Снайдер
Трент Хейн
Бэн Уэйли



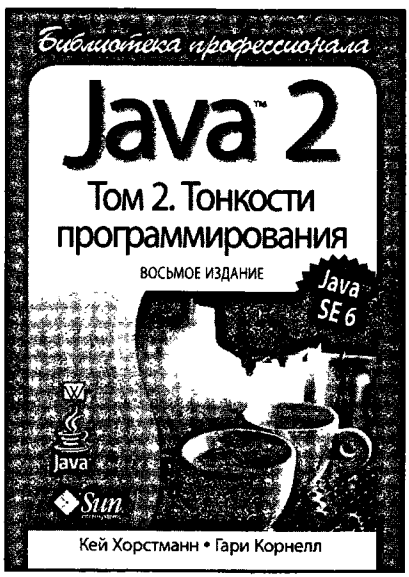
www.williamspublishing.com

В новом издании всемирно известной книги описываются эффективные методы работы и все аспекты управления системами UNIX и Linux, включая управление памятью, проектирование и управление сетями, электронную почту, веб-хостинг, создание сценариев, управление конфигурациями программного обеспечения, анализ производительности, взаимодействие с системой Windows, виртуализацию, DNS, безопасность, управление провайдерами IT-услуг и многое другое. В справочнике описаны современные версии систем UNIX и Linux — Solaris, HP-UX, AIX, Ubuntu Linux, openSUSE и Red Hat Enterprise Linux. Книга будет чрезвычайно полезной всем системным администраторам, а также пользователям систем UNIX и Linux, студентам, преподавателям и специалистам по сетевым технологиям.

ISBN 978-5-8459-1740-9 **в продаже**

JAVA™ 2
БИБЛИОТЕКА ПРОФЕССИОНАЛА
ТОМ 2. ТОНКОСТИ ПРОГРАММИРОВАНИЯ
ВОСЬМОЕ ИЗДАНИЕ

Кей Хорстманн
Гари Корнелл



www.williamspublishing.com

Книга ведущих специалистов по программированию на языке Java представляет собой обновленное издание фундаментального труда, учитывающее всю специфику новой версии платформы Java SE 6. Подробно рассматриваются такие темы, как новые средства ввода-вывода, использование XML, API-интерфейсы работы в сети и взаимодействия с базами данных (JDBC), интернационализация, построение графических интерфейсов, безопасность, JavaBeans, взаимодействие с распределенными объектами и кодом, написанным на языке сценариев, а также платформенно-ориентированные методы, позволяющие обращаться на низком уровне к функциям операционной системы. Книга изобилует множеством примеров, которые не только иллюстрируют концепции, но также демонстрируют способы правильной разработки, применяемые в реальных условиях.

ISBN 978-5-8459-1482-8 **в продаже**